# KodeKloud 50 Days AWS Cloud Challenge

# Table of Content

# Day 01: Create Key Pair

## Task Description:

The Nautilus DevOps team is strategizing the migration of a portion of their infrastructure to the AWS cloud. Recognizing the scale of this undertaking, they have opted to approach the migration in incremental steps rather than as a single massive transition. To achieve this, they have segmented large tasks into smaller, more manageable units. This granular approach enables the team to execute the migration in gradual phases, ensuring smoother implementation and minimizing disruption to ongoing operations. By breaking down the migration into smaller tasks, the Nautilus DevOps team can systematically progress through each stage, allowing for better control, risk mitigation, and optimization of resources throughout the migration process.
For this task, create a key pair with the following requirements:

- Name of the key pair should be **xfusion-kp**.
- Key pair type must be **rsa**

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***
Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI
If terminal is configured with AWS so we can create key pair through AWS CLI Command

```
aws ec2 create-key-pair --region us-east-1 --key-name xfusion-kp --key-type
rsa --query 'KeyMaterial' --output text > xfusion-kp.pem
```

- **create-key-pair** → Creates a new EC2 key pair
- **--region us-east-1** → Key pair will be available only in us-east-1
- **--key-name xfusion-kp** → Name of the key pair in AWS
- **--key-type rsa** → Uses RSA encryption algorithm
- **--query 'KeyMaterial'** → Extracts only the private key
- **--output text > xfusion-kp.pem** → Saves the private key to a file in a plain text format

**Change Key Permission**

```
chmod 600 xfusion-kp.pem
```

**To verify the key pair was created**

```
aws ec2 describe-key-pairs --region us-east-1 --key-name xfusion-kp
```

**List all key pairs**

```
aws ec2 describe-key-pairs --region us-east-1
```

Using AWS Console (Web UI)

1. Sign in to **AWS Management Console**
2. Navigate to **EC2** Dashboard
3. In the left sidebar, click "**Key Pairs**".
4. Click "**Create key pair**" button
5. Enter the following details:
    - Name: **xfusion-kp**
    - Key pair type: **RSA**
    - Private key format: **PEM** (for Linux/macOS)
6. Click "**Create key pair**"
7. The private key will automatically download - save it securely!

**Change Key Permission**

```
chmod 600 xfusion-kp.pem
```

# Day 02: Create Security Group

## Task Description:

The Nautilus DevOps team is strategizing the migration of a portion of their infrastructure to the AWS cloud. Recognizing the scale of this undertaking, they have opted to approach the migration in incremental steps rather than as a single massive transition. To achieve this, they have segmented large tasks into smaller, more manageable units. This granular approach enables the team to execute the migration in gradual phases, ensuring smoother implementation and minimizing disruption to ongoing operations. By breaking down the migration into smaller tasks, the Nautilus DevOps team can systematically progress through each stage, allowing for better control, risk mitigation, and optimization of resources throughout the migration process.

For this task, create a security group under default VPC with the following requirements:

- Name of the security group is `nautilus-sg`.
- The description must be `Security group for Nautilus App Servers`
- Add the inbound rule of type `HTTP`, with port range of `80`. Enter the source CIDR range of `0.0.0.0/0`.
- Add another inbound rule of type `SSH`, with port range of `22`. Enter the source CIDR range of `0.0.0.0/0`.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

First we need to get the **default VPC ID** of **us-east-1** region through aws cli

```
DEFAULT_VPC_ID=$(aws ec2 describe-vpcs --region us-east-1 --filters
"Name=isDefault,Values=true" --query "Vpcs[0].VpcId" --output text)

echo "Default VPC ID: $DEFAULT_VPC_ID"
```

**Create the security group**

```
SG_ID=$(aws ec2 create-security-group --group-name nautilus-sg
--description "Security group for Nautilus App Servers" --vpc-id
$DEFAULT_VPC_ID --region us-east-1 --query 'GroupId' --output text)

echo "Created Security Group ID: $SG_ID"
```

**Add HTTP inbound rule (port 80):**

```
aws ec2 authorize-security-group-ingress --group-id $SG_ID --protocol tcp
--port 80 --cidr 0.0.0.0/0 --region us-east-1
```

**Add HTTP inbound rule (port 22):**

```
aws ec2 authorize-security-group-ingress --group-id $SG_ID --protocol tcp
--port 22 --cidr 0.0.0.0/0 --region us-east-1
```

**Verification**

```
aws ec2 describe-security-groups --group-ids $SG_ID --query
'SecurityGroups[].IpPermissions'
```

# Day 03: Create Subnet

## Task Description:

The Nautilus DevOps team is strategizing the migration of a portion of their infrastructure to the AWS cloud. Recognizing the scale of this undertaking, they have opted to approach the migration in incremental steps rather than as a single massive transition.

For this task, create one subnet named `datacenter-subnet` under default VPC.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

First we need to Identify the Default VPC ID

```
DEFAULT_VPC_ID=$(aws ec2 describe-vpcs --region us-east-1 --filters
"Name=isDefault,Values=true" --query 'Vpcs[0].VpcId' --output text)

echo "Default VPC ID in us-east-1: $DEFAULT_VPC_ID"
```

**Determine an Available CIDR Block**

```
aws ec2 describe-subnets --region us-east-1 --filters
"Name=vpc-id,Values=$DEFAULT_VPC_ID" --query 'Subnets[].CidrBlock'
```

Assuming **172.31.96.0/20** is an available block within the default VPC's CIDR range.

Create the subnet with the chosen CIDR block, ensuring it is explicitly created in the us-east-1 region.

```
SUBNET_ID=$(aws ec2 create-subnet --region us-east-1 --vpc-id
$DEFAULT_VPC_ID --cidr-block 172.31.96.0/20 --tag-specifications
'ResourceType=subnet,Tags=[{Key=Name,Value=datacenter-subnet}]' --query
'Subnet.SubnetId' --output text)

echo "Created Subnet ID: $SUBNET_ID"
```

**Verification**

```
aws ec2 describe-subnets --region us-east-1 --subnet-ids $SUBNET_ID
```

# Day 04: Allocate Elastic IP

## Task Description:

The Nautilus DevOps team is strategizing the migration of a portion of their infrastructure to the AWS cloud. Recognizing the scale of this undertaking, they have opted to approach the migration in incremental steps rather than as a single massive transition.

For this task, allocate an `Elastic IP` address, name it as `nautilus-eip`

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

Command allocates a new Elastic IP address for use with a VPC and applies the required tag.

```
aws ec2 allocate-address --region us-east-1 --domain vpc
--tag-specifications
'ResourceType=elastic-ip,Tags=[{Key=Name,Value=nautilus-eip}]' --query
'PublicIp' --output text
```

# Day 05: Create GP3 Volume

## Task Description:

The Nautilus DevOps team is strategizing the migration of a portion of their infrastructure to the AWS cloud. Recognizing the scale of this undertaking, they have opted to approach the migration in incremental steps rather than as a single massive transition.
Create a volume with the following requirements:

- Name of the volume should be `datacenter-volume`.
- Volume `type` must be `gp3`.
- Volume `size` must be `2 GiB`.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)
***Notes:***
Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Creating the EBS Volume**

```
aws ec2 create-volume --region us-east-1 --volume-type gp3 --size 2
--availability-zone us-east-1a --tag-specifications
'ResourceType=volume,Tags=[{Key=Name,Value=datacenter-volume}]'
```

**Validation**

```
aws ec2 describe-volumes --region us-east-1 --filters
"Name=tag:Name,Values=datacenter-volume" --query "Volumes[*].{ID:VolumeId,
```

```
State:State, Type:VolumeType, Size_GiB:Size, AZ:AvailabilityZone,
Tags:Tags[?Key=='Name']|[0].Value}" --output table
```

# Day 06: Launch EC2 Instance

## Task Description:

The Nautilus DevOps team is strategizing the migration of a portion of their infrastructure to the AWS cloud. Recognizing the scale of this undertaking, they have opted to approach the migration in incremental steps rather than as a single massive transition. To achieve this, they have segmented large tasks into smaller, more manageable units.

For this task, create an EC2 instance with following requirements:

1. The name of the instance must be `xfusion-ec2`.
2. You can use the `Amazon Linux` AMI to launch this instance.
3. The Instance type must be `t2.micro`.
4. Create a new RSA key pair named `xfusion-kp`.
5. Attach the default (available by default) security group.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

If terminal is configured with AWS so we can create key pair through AWS CLI Command

```
aws ec2 create-key-pair --region us-east-1 --key-name xfusion-kp --key-type
```

```
rsa --query 'KeyMaterial' --output text > xfusion-kp.pem
```

**To verify the key pair was created**

```
aws ec2 describe-key-pairs --region us-east-1 --key-name xfusion-kp
```

**Change Key Permission**

```
chmod 600 xfusion-kp.pem
```

**Retrieve the Latest AMI ID using SSM Parameter Store**

```
AMI_ID=$(aws ssm get-parameter --region us-east-1 --name
/aws/service/ami-amazon-linux-latest/al2023-ami-kernel-default-x86_64
--query 'Parameter.Value' --output text)

echo "Using AMI ID: $AMI_ID"
```

**Launch the EC2 Instance**

```
aws ec2 run-instances --region us-east-1 --image-id $AMI_ID --instance-type
t2.micro --key-name xfusion-kp --count 1 --tag-specifications
'ResourceType=instance,Tags=[{Key=Name,Value=xfusion-ec2}]' --query
'Instances[0].InstanceId' --output text
```

Since we did not pass the Subnet ID and Security group ID in the EC2 creation command, the reason for this was, If we did not mention the subnet ID and Security group ID it would pick the default VPC and Security group in that Region

**AWS Default VPC/Subnet Rule**

**Verification**

```
aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=xfusion-ec2" --query "Reservations[].Instances[].{
ID:InstanceId, State:State.Name, Type:InstanceType, KeyName:KeyName,
PublicIP:PublicIpAddress, VPC:VpcId, Subnet:SubnetId,
SecurityGroups:join(', ', NetworkInterfaces[].Groups[].GroupId) }" --output
table
```

# Day 07: Change EC2 Instance Type

## Task Description:

During the migration process, the Nautilus DevOps team created several EC2 instances in different regions. They are currently in the process of identifying the correct resources and utilization and are making continuous changes to ensure optimal resource utilization. Recently, they discovered that one of the EC2 instances was underutilized, prompting them to decide to change the instance type. Please make sure the `Status check` is completed (if its still in `Initializing` state) before making any changes to the instance.

1) Change the instance type from `t2.micro` to `t2.nano` for `nautilus-ec2` instance.

2) Make sure the ec2 instance `nautilus-ec2` is in `running` state after the change.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

Get the Instance ID for `nautilus-ec2` using its Name tag.

```
INSTANCE_ID=$(aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=nautilus-ec2" --query
'Reservations[].Instances[0].InstanceId' --output text)
```

```
echo "Instance ID for nautilus-ec2: $INSTANCE_ID"
```

**Check the Instance Status**

```
aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=nautilus-ec2" --query
"Reservations[].Instances[0].{Name:Tags[?Key=='Name']|[0].Value,
State:State.Name}" --output table
```

We cannot change the instance type while the instance is running. So we need to stop the running instance first.

```
echo "Stopping the instance $INSTANCE_ID..."
aws ec2 stop-instances --region us-east-1 --instance-ids $INSTANCE_ID
```

Wait for the instance state to change to 'stopped'

**Check the Instance Status**

```
aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=nautilus-ec2" --query
"Reservations[].Instances[0].{Name:Tags[?Key=='Name']|[0].Value,
State:State.Name}" --output table
```

**Change the Instance Type (t2.micro to t2.nano)**

```
echo "Changing instance type to t2.nano..."

aws ec2 modify-instance-attribute --region us-east-1 --instance-id
$INSTANCE_ID --instance-type "t2.nano
```

**Check the Instance Type and Status**

```
aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=nautilus-ec2" --query
"Reservations[].Instances[0].{Name:Tags[?Key=='Name']|[0].Value,
State:State.Name, Type:InstanceType}" --output table
```

**Start the Instance**

```
echo "Starting the instance $INSTANCE_ID..."
aws ec2 start-instances --region us-east-1 --instance-ids $INSTANCE_ID
```

# Day 08: Enable Stop Protection for EC2 Instance

## Task Description:

As part of the migration, there were some components added to the AWS account. Team created one of the EC2 instances where they need to make some changes now.

There is an EC2 instance named `datacenter-ec2` under `us-east-1` region, enable the `stop` protection for this instance.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)
***Notes:***
Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

Get the Instance ID for `datacenter-ec2` using its Name tag.

```
INSTANCE_ID=$(aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=datacenter-ec2" --query
'Reservations[].Instances[0].InstanceId' --output text)

echo "Instance ID for datacenter-ec2: $INSTANCE_ID"
```

**Modify Instance Attribute to Enable Stop Protection**

```
echo "Enabling Stop Protection for $INSTANCE_ID..."
aws ec2 modify-instance-attribute --region us-east-1 --instance-id
$INSTANCE_ID --disable-api-stop
```

**Verification**

```
aws ec2 describe-instance-attribute --region us-east-1 --instance-id
$INSTANCE_ID --attribute disableApiStop --query 'DisableApiStop.Value'
--output text
```

The successful output of the verification command will be: **True**

# Day 09: Enable Termination Protection for EC2 Instance

## Task Description:

As part of the migration, there were some components created under the AWS account. The Nautilus DevOps team created one EC2 instance where they forgot to enable the termination protection which is needed for this instance.

An instance named `nautilus-ec2` already exists in `us-east-1` region. Enable `termination protection` for the same.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)
***Notes:***
Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

Get the Instance ID for `nautilus-ec2` using its Name tag.

```
INSTANCE_ID=$(aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=nautilus-ec2" --query
'Reservations[].Instances[0].InstanceId' --output text)
```

```
echo "Instance ID for datacenter-ec2: $INSTANCE_ID"
```

**Modify Instance Attribute to Enable Termination Protection**

```
echo "Enabling Termination Protection for $INSTANCE_ID..."

aws ec2 modify-instance-attribute --region us-east-1 --instance-id
$INSTANCE_ID --disable-api-termination
```

**Verification**

```
aws ec2 describe-instance-attribute --region us-east-1 --instance-id
$INSTANCE_ID --attribute disableApiTermination --query
'DisableApiTermination.Value' --output text
```

The successful output of the verification command will be: **True**

# Day 10: Attach Elastic IP to EC2 Instance

## Task Description:

The Nautilus DevOps team has been creating a couple of services on AWS cloud. They have been breaking down the migration into smaller tasks, allowing for better control, risk mitigation, and optimization of resources throughout the migration process. Recently they came up with requirements mentioned below.

There is an instance named `nautilus-ec2` and an elastic-ip named `nautilus-ec2-eip` in `us-east-1` region. Attach the `nautilus-ec2-eip` elastic-ip to the `nautilus-ec2` instance.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

Get the Instance ID for `nautilus-ec2` using its Name tag.

```
INSTANCE_ID=$(aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=nautilus-ec2" --query
'Reservations[].Instances[0].InstanceId' --output text)

echo "Instance ID for datacenter-ec2: $INSTANCE_ID"
```

**Create Elastic IP**

Command allocates a new Elastic IP address for use with a VPC and applies the required tag.

```
aws ec2 allocate-address --region us-east-1 --domain vpc
--tag-specifications
'ResourceType=elastic-ip,Tags=[{Key=Name,Value=nautilus-ec2-eip}]' --query
'PublicIp' --output text
```

Get the Allocation ID and Public IP of nautilus-ec2-eip

```
EIP_DETAILS=$(aws ec2 describe-addresses --region us-east-1 --filters
"Name=tag:Name,Values=nautilus-ec2-eip" --query
'Addresses[0].[AllocationId, PublicIp]' --output text)

ALLOCATION_ID=$(echo $EIP_DETAILS | awk '{print $1}')
PUBLIC_IP=$(echo $EIP_DETAILS | awk '{print $2}')

echo "Allocation ID: $ALLOCATION_ID"
echo "Public IP: $PUBLIC_IP"
```

**Associate the Elastic IP**

```
echo "Attaching Elastic IP ($PUBLIC_IP) to Instance ($INSTANCE_ID)..."

aws ec2 associate-address --region us-east-1 --instance-id $INSTANCE_ID
--allocation-id $ALLOCATION_ID
```

**Verification**

```
echo "Verifying new public IP address..."
aws ec2 describe-instances --region us-east-1 --instance-ids $INSTANCE_ID
--query 'Reservations[].Instances[0].PublicIpAddress' --output text
```

# Day 11: Attach Elastic Network Interface to EC2 Instance

## Task Description:

The Nautilus DevOps team has been creating a couple of services on AWS cloud. They have been breaking down the migration into smaller tasks, allowing for better control, risk mitigation, and optimization of resources throughout the migration process. Recently they came up with requirements mentioned below.

An instance named `nautilus-ec2` and an elastic network interface named `nautilus-eni` already exists in `us-east-1` region.

- Please make sure instance initialisation has been completed before submitting this task.
- Attach the `nautilus-eni` network interface to the `nautilus-ec2` instance.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Identify Instance ID**

```
INSTANCE_ID=$(aws ec2 describe-instances --region us-east-1 --filters
```

```
"Name=tag:Name,Values=nautilus-ec2" --query
'Reservations[].Instances[0].InstanceId' --output text)

echo "Instance ID: $INSTANCE_ID"
```

**Get the Network Interface ID**

```
ENI_ID=$(aws ec2 describe-network-interfaces --region us-east-1 --filters
"Name=tag:Name,Values=nautilus-eni" --query
'NetworkInterfaces[0].NetworkInterfaceId' --output text)

echo "Network Interface ID: $ENI_ID"
```

**Attach the Network Interface**

```
ATTACHMENT_ID=$(aws ec2 attach-network-interface --region us-east-1
--instance-id $INSTANCE_ID --network-interface-id $ENI_ID --device-index 1
--query 'AttachmentId' --output text)

echo "Attachment ID: $ATTACHMENT_ID"
```

**Verification**

```
aws ec2 describe-instances --region us-east-1 --instance-ids $INSTANCE_ID
--query
"Reservations[].Instances[].NetworkInterfaces[?Attachment.NetworkInterfaceI
d=='$ENI_ID'].Attachment.DeviceIndex" --output text
```

# Day 12: Attach Volume to EC2 Instance

## Task Description:

The Nautilus DevOps team has been creating a couple of services on AWS cloud. They have been breaking down the migration into smaller tasks, allowing for better control, risk mitigation, and optimization of resources throughout the migration process. Recently they came up with requirements mentioned below.

An instance named `xfusion-ec2` and a volume named `xfusion-volume` already exists in `us-east-1` region. Attach the `xfusion-volume` volume to the `xfusion-ec2` instance, make sure to set the device name to `/dev/sdb` while attaching the volume.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Identify Instance ID**

```
INSTANCE_ID=$(aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=xfusion-ec2" --query
"Reservations[].Instances[0].InstanceId" --output text)
```

```
echo "Instance ID: $INSTANCE_ID"
```

**Get the Volume ID**

```
VOLUME_ID=$(aws ec2 describe-volumes --region us-east-1 --filters
"Name=tag:Name,Values=xfusion-volume" --query "Volumes[0].VolumeId"
--output text)

echo "Volume ID: $VOLUME_ID"
```

**Attach the Volume**

```
aws ec2 attach-volume --region us-east-1 --volume-id $VOLUME_ID
--instance-id $INSTANCE_ID --device /dev/sdb
```

**Verify**

```
aws ec2 describe-volumes --region us-east-1 --volume-ids $VOLUME_ID --query
"Volumes[0].Attachments[0].{Instance:InstanceId,Device:Device,Status:State}
" --output table
```

# Day 13: Create AMI from EC2 Instance

## Task Description:

The Nautilus DevOps team is strategizing the migration of a portion of their infrastructure to the AWS cloud. Recognizing the scale of this undertaking, they have opted to approach the migration in incremental steps rather than as a single massive transition. To achieve this, they have segmented large tasks into smaller, more manageable units.

For this task, create an AMI from an existing EC2 instance named `xfusion-ec2` with the following requirement:

Name of the AMI should be `xfusion-ec2-ami`, make sure AMI is in `available` state.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Identify Instance ID**

```
INSTANCE_ID=$(aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=xfusion-ec2" --query
"Reservations[].Instances[0].InstanceId" --output text)
```

```
echo "Instance ID: $INSTANCE_ID"
```

**Create the AMI**
Run the **`create-image`** command. By default, AWS will stop and restart the instance to ensure data consistency during the image creation.

```
AMI_ID=$(aws ec2 create-image --region us-east-1 --instance-id $INSTANCE_ID
--name "xfusion-ec2-ami" --description "Migration image for xfusion-ec2"
--query "ImageId" --output text)

echo "New AMI ID: $AMI_ID"
```

**Wait for the AMI to be Available**
Use the wait command. This will pause the terminal until the AMI transitions from pending to available.

```
echo "Waiting for AMI to become available..."
aws ec2 wait image-available --region us-east-1 --image-ids $AMI_ID

echo "Success! AMI $AMI_ID is now available."
```

**AWS Management Console (GUI)**
Follow these steps to create the image via the web interface:

1. Log in to the AWS Management Console and ensure our region is set to US East (N. Virginia) **us-east-1**.
2. Navigate to **EC2 > Instances > Instances**.
3. Find and select the checkbox for the instance named **xfusion-ec2**.
4. Click the Actions button at the top right, then select **Image and templates > Create image**.
5. On the Create image page:
    - Image name: Enter **xfusion-ec2-ami**.
    - Image description: (Optional) "**AMI for xfusion-ec2 migration**".
    - Leave other settings (Reboot, Instance volumes) as default unless specifically required.
6. Click Create image at the bottom of the page.
7. Monitor the Status:
    - In the left-hand navigation pane, under Images, click AMIs.
    - Locate **xfusion-ec2-ami**. we will see the Status column showing pending. Once it changes to available, the task is complete.

# Day 14: Terminate EC2 Instance

## Task Description:

During the migration process, several resources were created under the AWS account. Later on, some of these resources became obsolete as alternative solutions were implemented. Similarly, there is an instance that needs to be deleted as it is no longer in use.

1) Delete the ec2 instance named `xfusion-ec2` present in `us-east-1` region.

2) Before submitting your task, make sure instance is in `terminated` state.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Identify Instance ID**

```
INSTANCE_ID=$(aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=xfusion-ec2" --query
"Reservations[].Instances[0].InstanceId" --output text)

echo "Instance ID: $INSTANCE_ID"
```

**Check Termination Protection**

```
aws ec2 describe-instance-attribute --region us-east-1 --instance-id
$INSTANCE_ID --attribute disableApiTermination --query
"DisableApiTermination.Value" --output text
```

**True**: Termination Protection is ENABLED.
**False:** Termination Protection is DISABLED.

**Disable Termination Protection (If Enabled)**

```
aws ec2 modify-instance-attribute --region us-east-1 --instance-id
$INSTANCE_ID --no-disable-api-termination
```

**Terminate the Instance**

Execute the termination. This will immediately change the instance state to shutting-down.

```
aws ec2 terminate-instances --region us-east-1 --instance-ids $INSTANCE_ID
```

**Verify**

```
aws ec2 describe-instances --region us-east-1 --instance-ids $INSTANCE_ID
--query "Reservations[].Instances[].{ID:InstanceId,State:State.Name}"
--output table
```

# Day 15: Create Volume Snapshot

## Task Description:

The Nautilus DevOps team has some volumes in different regions in their AWS account. They are going to setup some automated backups so that all important data can be backed up on regular basis. For now they shared some requirements to take a snapshot of one of the volumes they have.

Create a snapshot of an existing volume named `nautilus-vol` in `us-east-1` region.

1) The name of the snapshot must be `nautilus-vol-ss`.
2) The description must be `nautilus Snapshot`.
3) Make sure the snapshot status is `completed` before submitting the task.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Get the Volume ID**

```
VOLUME_ID=$(aws ec2 describe-volumes --region us-east-1 --filters
"Name=tag:Name,Values=nautilus-vol" --query "Volumes[0].VolumeId" --output
```

```
text)

echo "Volume ID: $VOLUME_ID"
```

**Create the Snapshot**

```
SNAPSHOT_ID=$(aws ec2 create-snapshot --region us-east-1 --volume-id
$VOLUME_ID --description "nautilus Snapshot" --tag-specifications
"ResourceType=snapshot,Tags=[{Key=Name,Value=nautilus-vol-ss}]" --query
"SnapshotId" --output text)

echo "Created Snapshot ID: $SNAPSHOT_ID"
```

**Verify**

```
aws ec2 describe-snapshots --region us-east-1 --snapshot-ids $SNAPSHOT_ID
--query "Snapshots[].{ID:SnapshotId, Name:Tags[?Key=='Name']|[0].Value,
Description:Description, Status:State}" --output table
```

# Day 16: Create IAM User

## Task Description:

When establishing infrastructure on the AWS cloud, Identity and Access Management (IAM) is among the first and most critical services to configure. IAM facilitates the creation and management of user accounts, groups, roles, policies, and other access controls. The Nautilus DevOps team is currently in the process of configuring these resources and has outlined the following requirements:

For this task, create an IAM user named `iamuser_jim`.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create the IAM User**

```
aws iam create-user --user-name iamuser_jim --region us-east-1
```

**Verification**

```
aws iam get-user --user-name iamuser_jim --query 'User.{UserId:UserId,
UserName:UserName, Arn:Arn}' --output table
```

# Day 17: Create IAM Group

## Task Description:

The Nautilus DevOps team has been creating a couple of services on AWS cloud. They have been breaking down the migration into smaller tasks, allowing for better control, risk mitigation, and optimization of resources throughout the migration process. Recently they came up with requirements mentioned below.

Create an IAM group named `iamgroup_kirsty`.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)
***Notes:***
Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI
**Create the IAM Group**

```
aws iam create-group --group-name iamgroup_kirsty
```

**Verification**

```
aws iam get-group --group-name iamgroup_kirsty --query
'Group.{GroupName:GroupName, GroupId:GroupId, Arn:Arn}' --output table
```

# Day 18: Create Read-Only IAM Policy for EC2 Console Access

## Task Description:

When establishing infrastructure on the AWS cloud, Identity and Access Management (IAM) is among the first and most critical services to configure. The Nautilus DevOps team is currently in the process of configuring these resources and has outlined the following requirements.

Create an IAM policy named `iampolicy_kareem` in `us-east-1` region, it must allow read-only access to the EC2 console, i.e this policy must allow users to view all instances, AMIs, and snapshots in the Amazon EC2 console.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Policy Definition (JSON)**

To view instances, AMIs, and snapshots in the console, the user needs "Describe" permissions.

**Create policy JSON file in the local Machine**

```
touch iampolicy_kareem.json
```

Add the following rule in the file

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeInstances",
                "ec2:DescribeImages",
                "ec2:DescribeSnapshots",
                "ec2:DescribeTags",
                "ec2:DescribeVolumes"
            ],
            "Resource": "*"
        }
    ]
}
```

**Create the IAM Policy**

```
aws iam create-policy --policy-name iampolicy_kareem --policy-document
file://iampolicy_kareem.json --description "Read-only access to EC2
instances, AMIs, and snapshots" --region us-east-1
```

**Verification**

```
aws iam list-policies --scope Local --query
"Policies[?PolicyName=='iampolicy_kareem'].{Name:PolicyName, ARN:Arn}"
--output table
```

**AWS Management Console (GUI)**

1. Log in to the AWS Console and navigate to IAM.
2. In the left sidebar, click Policies, then click Create policy.
3. On the Specify permissions page, click the JSON tab.
4. Paste the **JSON code provided above** into the editor, replacing the existing text.
5. Click Next.
6. On the Review and create page:
    - Policy name: iampolicy_kareem
    - Description: "Allows read-only access to EC2 console resources."
7. Click Create policy.

# Day 19: Attach IAM Policy to IAM User

## Task Description:

The Nautilus DevOps team has been creating a couple of services on AWS cloud. They have been breaking down the migration into smaller tasks, allowing for better control, risk mitigation, and optimization of resources throughout the migration process. Recently they came up with requirements mentioned below.

An IAM user named `iamuser_james` and a policy named `iampolicy_james` already exist. Attach the IAM policy `iampolicy_james` to the IAM user `iamuser_james`.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)
***Notes:***
Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Retrieve the Policy ARN**

```
POLICY_ARN=$(aws iam list-policies --scope Local --query
"Policies[?PolicyName=='iampolicy_james'].Arn" --output text)

echo "Policy ARN: $POLICY_ARN"
```

**Attach the Policy to the User**

```
aws iam attach-user-policy --user-name iamuser_james --policy-arn
$POLICY_ARN
```

**Verification**

```
aws iam list-attached-user-policies --user-name iamuser_james --query
"AttachedPolicies[?PolicyName=='iampolicy_james']" --output table
```

**AWS Management Console (GUI)**
If you prefer to use the visual interface, follow these steps:

1. **Log in** to the AWS Management Console and navigate to the **IAM** dashboard.
2. In the left-hand navigation pane, click **Users**.
3. Find and click on the username **`iamuser_james`**.
4. In the **Permissions** tab, click the **Add permissions** dropdown menu and select **Add permissions**.
5. Select **Attach policies directly**.
6. In the **Permissions policies** search box, type `iampolicy_james`.
7. Check the box next to the policy name in the search results.
8. Click **Next** at the bottom of the page.
9. Review the details and click **Add permissions**.

# Day 20: Create IAM Role for EC2 with Policy Attachment

## Task Description:

When establishing infrastructure on the AWS cloud, Identity and Access Management (IAM) is among the first and most critical services to configure. The Nautilus DevOps team is currently in the process of configuring these resources and has outlined the following requirements:

Create an IAM role as below:

1. IAM role name must be **iamrole_yousuf**
2. Entity type must be AWS Service and use case must be EC2.
3. Attach a policy named **iampolicy_yousuf**

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create the Trust Policy Document**

create a local JSON file that defines the trust relationship for the EC2 service.

```
touch trust-policy.json
```

Add the following content in the file

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "ec2.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

**Create the IAM Role**

Use the `create-role` command and point it to the file we just created.

```
aws iam create-role --role-name iamrole_yousuf
--assume-role-policy-document file://trust-policy.json --description "Role
for EC2 to access resources via iampolicy_yousuf"
```

**Attach the Policy**

First, retrieve the **ARN** for the existing `iampolicy_yousuf`, then attach it to the new role.

Get the Policy ARN

```
POLICY_ARN=$(aws iam list-policies --scope Local --query
"Policies[?PolicyName=='iampolicy_yousuf'].Arn" --output text)

echo $POLICY_ARN
```

**Attach the policy to the role**

```
aws iam attach-role-policy --role-name iamrole_yousuf --policy-arn
$POLICY_ARN
```

**Verification**

```
aws iam list-attached-role-policies --role-name iamrole_yousuf --output
table
```

# Day 21: Setting Up an EC2 Instance with an Elastic IP for Application Hosting

## Task Description:

The Nautilus DevOps Team has received a new request from the Development Team to set up a new EC2 instance. This instance will be used to host a new application that requires a stable IP address. To ensure that the instance has a consistent public IP, an Elastic IP address needs to be associated with it. The instance will be named `devops-ec2`, and the Elastic IP will be named `devops-eip`. This setup will help the Development Team to have a reliable and consistent access point for their application.

Create an EC2 instance named `devops-ec2` using any linux AMI like ubuntu, the Instance type must be `t2.micro` and associate an `Elastic IP` address with this instance, name it as `devops-eip`.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)
***Notes:***
Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create key pair**

```
aws ec2 create-key-pair --region us-east-1 --key-name devops-kp --key-type
rsa --query 'KeyMaterial' --output text > devops-kp.pem
```

**To verify the key pair was created**

```
aws ec2 describe-key-pairs --region us-east-1 --key-name devops-kp
```

**Change Key Permission**

```
chmod 600 devops-kp.pem
```

**Retrieve the Latest AMI ID using SSM Parameter Store**

```
UBUNTU_AMI_ID=$(aws ssm get-parameter --region us-east-1 --name
/aws/service/canonical/ubuntu/server/noble/stable/current/amd64/hvm/ebs-gp3
/ami-id --query 'Parameter.Value' --output text)


echo "Using AMI ID: $UBUNTU_AMI_ID"
```

**Launch the EC2 Instance**

```
INSTANCE_ID=$(aws ec2 run-instances --region us-east-1 --image-id
$UBUNTU_AMI_ID --instance-type t2.micro --key-name devops-kp --count 1
--tag-specifications
'ResourceType=instance,Tags=[{Key=Name,Value=devops-ec2}]' --query
'Instances[0].InstanceId' --output text)

echo "Launched Instance ID: $INSTANCE_ID"
```

Since we did not pass the Subnet ID and Security group ID in the EC2 creation command, the reason for this was, If we did not mention the subnet ID and Security group ID it would pick the default VPC and Security group in that Region

**AWS Default VPC/Subnet Rule**

**Verification**

```
aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=devops-ec2" --query "Reservations[].Instances[].{
ID:InstanceId, State:State.Name, Type:InstanceType, KeyName:KeyName,
PublicIP:PublicIpAddress, VPC:VpcId, Subnet:SubnetId,
SecurityGroups:join(', ', NetworkInterfaces[].Groups[].GroupId) }" --output
table
```

**Allocate and Name the Elastic IP**

```
ALLOCATION_ID=$(aws ec2 allocate-address --region us-east-1
--tag-specifications
'ResourceType=elastic-ip,Tags=[{Key=Name,Value=devops-eip}]' --query
'AllocationId' --output text)

echo "Allocated EIP ID: $ALLOCATION_ID"
```

**Get the Elastic IP**

```
aws ec2 describe-addresses --region us-east-1 --filters
"Name=tag:Name,Values=devops-eip" --query
"Addresses[*].{Name:Tags[?Key=='Name']|[0].Value, IP:PublicIp,
Allocation:AllocationId, Instance:InstanceId}" --output table
```

**Associate the Elastic IP with the Instance**

```
aws ec2 associate-address --region us-east-1 --instance-id $INSTANCE_ID
--allocation-id $ALLOCATION_ID

echo "Success! Elastic IP is now associated with devops-ec2."
```

# Day 22: Configuring Secure SSH Access to an EC2 Instance

## Task Description:

The Nautilus DevOps team needs to set up a new EC2 instance that can be accessed securely from their landing host (`aws-client`). The instance should be of type `t2.micro` and named `xfusion-ec2`. A new SSH key should be created on the `aws-client` host under the`/root/.ssh/` folder, if it doesn't already exist. This key should then be added to the `root` user's authorised keys on the EC2 instance, allowing passwordless SSH access from the `aws-client` host.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

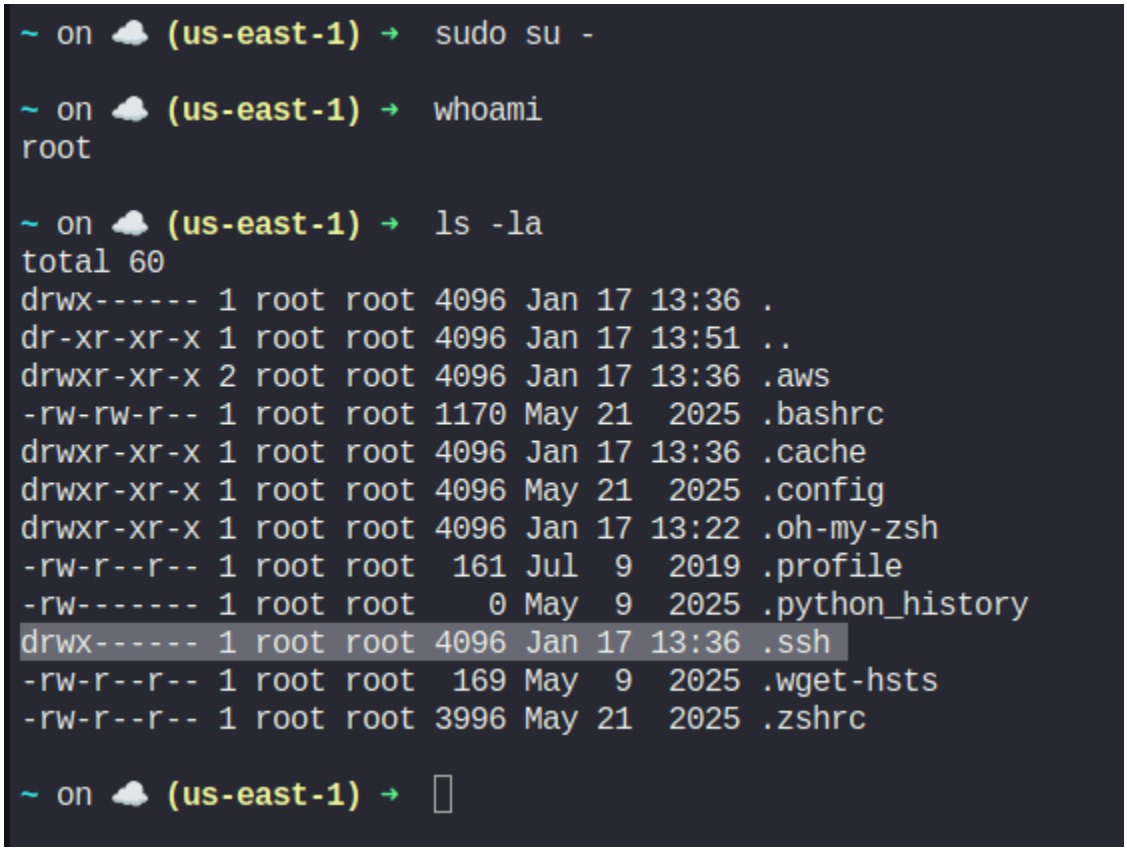
**Generate the SSH Key on `aws-client`**

Create the SSH directory, change the permission in **root** user in the **aws-client** machine.

```
# Switch to root user
sudo su -
```

```
# Create SSH directory in root user
mkdir ~/.ssh

# Change the Permission of directory
chmod 700 ~/.ssh
```




Generate the SSH Key

```
ssh-keygen -t rsa -b 2048
```

```
~ on ☁️  (us-east-1) ➜  cd .ssh

~/.ssh on ☁️  (us-east-1) ➜  pwd
/root/.ssh

~/.ssh on ☁️  (us-east-1) ➜  ssh-keygen -t rsa -b 2048
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa
Your public key has been saved in /root/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:RO9CQeDsvY41RYP5gkZPpXpzUh1i+QOzG8UXYziBGkY root@aws-client
The key's randomart image is:
+---[RSA 2048]----+
|      .+E ++oo+. |
|     o .oO=o=o.. |
|      +.Bo=*.o   |
|     o O.=o.o    |
|      = S =o .   |
|     . . O.      |
|        +        |
|       + .       |
|      . .        |
+----[SHA256]-----+

~/.ssh on ☁️  (us-east-1) ➜  ls
agent-environment  authorized_keys  id_rsa  id_rsa.pub
```

**Import the Public Key to AWS**

```
aws ec2 import-key-pair --region us-east-1 --key-name xfusion-kp
--public-key-material fileb:///root/.ssh/id_rsa.pub
```

**To verify the key pair was imported or not**

```
aws ec2 describe-key-pairs --region us-east-1 --key-name xfusion-kp
```

**Prepare the Configuration Script (User Data)**

Create the script file

```
touch userdata.sh
chmod +x userdata.sh
```

Add the following content in userdata.sh file

```
#!/bin/bash

# Setup Root SSH Directory
mkdir -p /root/.ssh
chmod 700 /root/.ssh

echo "<public key of user>" >> ~/.ssh/authorized_keys

chmod 600 /root/.ssh/authorized_keys
chown root:root /root/.ssh/authorized_keys

# SSH Hardening

# Set Root Login to key-only
sed -i 's/^#*PermitRootLogin.*/PermitRootLogin prohibit-password/g'
/etc/ssh/sshd_config

# Disable Password Authentication entirely
sed -i 's/^#*PasswordAuthentication.*/PasswordAuthentication no/g'
/etc/ssh/sshd_config # Disable Challenge-Response (Keyboard Interactive)

sed -i
's/^#*ChallengeResponseAuthentication.*/ChallengeResponseAuthentication
no/g' /etc/ssh/sshd_config

# Apply Changes
systemctl restart sshd
```

**Retrieve the Latest AMI ID using SSM Parameter Store**

```
AMI_ID=$(aws ssm get-parameter --region us-east-1 --name
/aws/service/ami-amazon-linux-latest/al2023-ami-kernel-default-x86_64
--query 'Parameter.Value' --output text)

echo "Using AMI ID: $AMI_ID"
```

**Launch the `xfusion-ec2` Instance**

```
aws ec2 run-instances --region us-east-1 --image-id $AMI_ID --count 1
--instance-type t2.micro --key-name xfusion-kp --user-data
file://userdata.sh --tag-specifications
'ResourceType=instance,Tags=[{Key=Name,Value=xfusion-ec2}]'
```

**Verify**

Get the Public IP of instance

```
PUBLIC_IP=$(aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=xfusion-ec2"
"Name=instance-state-name,Values=running" --query
"Reservations[0].Instances[0].PublicIpAddress" --output text)
```

SSH into the instance as root

```
ssh -i /root/.ssh/id_rsa root@$PUBLIC_IP
```

If the SSH connection show the timeout error please check the **Security Group inbound rules**

# Day 23: Data Migration Between S3 Buckets Using AWS CLI

## Task Description:

As part of a data migration project, the team lead has tasked the team with migrating data from an existing S3 bucket to a new S3 bucket. The existing bucket contains a substantial amount of data that must be accurately transferred to the new bucket. The team is responsible for creating the new S3 bucket and ensuring that all data from the existing bucket is copied or synced to the new bucket completely and accurately. It is imperative to perform thorough verification steps to confirm that all data has been successfully transferred to the new bucket without any loss or corruption.
As a member of the Nautilus DevOps Team, your task is to perform the following:

- **Create a New Private S3 Bucket:** Name the bucket `datacenter-sync-15239`.
- **Data Migration:** Migrate the entire data from the existing `datacenter-s3-29198` bucket to the new `datacenter-sync-15239` bucket.
- **Ensure Data Consistency:** Ensure that both buckets have the same data.
- **Use AWS CLI:** Use the AWS CLI to perform the creation and data migration tasks.

***Notes:***
Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create New Private S3 Bucket**
In AWS, buckets are private by default upon creation. However, best practice is to explicitly enable "**Block Public Access**" to ensure no data is accidentally exposed.

```
aws s3 mb s3://datacenter-sync-15239 --region us-east-1
```

Enforce the "Private" requirement by blocking all public access

```
aws s3api put-public-access-block --bucket datacenter-sync-15239
--public-access-block-configuration
"BlockPublicAcls=true,IgnorePublicAcls=true,BlockPublicPolicy=true,Restrict
PublicBuckets=true"
```

**Verify**

```
Aws s3 ls | grep datacenter-sync-15239
```

**Perform the Data Migration**
We will use the `sync` command instead of `cp`. The `sync` command is more robust because it automatically compares the source and destination and only copies files that are missing or have different sizes/timestamps.

```
aws s3 sync s3://datacenter-s3-29198 s3://datacenter-sync-15239 --delete
```

**`--delete` flag:** This flag ensures that if there are any stray files in the *destination* bucket that do not exist in the *source*, they are removed. This ensures the destination is a perfect mirror of the original.

**Verify Data Consistency**
To fulfill the requirement of "thorough verification," we compare the **Object Count** and **Total Size** of both buckets.

Check the Source Bucket:

```
aws s3 ls s3://datacenter-s3-29198 --recursive --summarize | tail -n 2
```

Check the Destination Bucket:

```
aws s3 ls s3://datacenter-sync-15239 --recursive --summarize | tail -n 2
```

# Day 24: Setting Up an Application Load Balancer for an EC2 Instance

## Task Description:

The Nautilus DevOps team is currently working on setting up a simple application on the AWS cloud. They aim to establish an Application Load Balancer (ALB) in front of an EC2 instance where an Nginx server is currently running. While the Nginx server currently serves a sample page, the team plans to deploy the actual application later.

1. Set up an Application Load Balancer named `xfusion-alb`.
2. Create a target group named `xfusion-tg`.
3. Create a security group named `xfusion-sg` to open port `80` for the public.
4. Attach this security group to the ALB.
5. The ALB should route traffic on port `80` to port `80` of the `xfusion-ec2` instance.
6. Make appropriate changes in the default security group attached to the EC2 instance if necessary.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Get the VPC ID and Subnet IDs**
we need the **VPC ID** and the **Subnet IDs**. An ALB requires **at least two subnets in different Availability Zones**.

Get VPC ID

```
VPC_ID=$(aws ec2 describe-vpcs --region us-east-1 --query 'Vpcs[0].VpcId'
--output text)

echo $VPC_ID
```

Get two subnets from different AZs

```
# Explicitly get one subnet from us-east-1a and one from us-east-1b

SUB_A=$(aws ec2 describe-subnets --region us-east-1 --filters
"Name=vpc-id,Values=$VPC_ID" "Name=availability-zone,Values=us-east-1a"
--query 'Subnets[0].SubnetId' --output text)

SUB_B=$(aws ec2 describe-subnets --region us-east-1 --filters
"Name=vpc-id,Values=$VPC_ID" "Name=availability-zone,Values=us-east-1b"
--query 'Subnets[0].SubnetId' --output text)

echo "Using Subnets: $SUB_A (1a) and $SUB_B (1b)"
```

**Get the Instance ID of xfusion-ec2**

```
INSTANCE_ID=$(aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=xfusion-ec2"
"Name=instance-state-name,Values=running" --query
"Reservations[0].Instances[0].InstanceId" --output text)

echo $INSTANCE_ID
```

Check the Instance's availability Zone

```
aws ec2 describe-instances --instance-ids $INSTANCE_ID --region us-east-1
--query
'Reservations[0].Instances[0].{Subnet:SubnetId,AZ:Placement.AvailabilityZon
e}'
```

If the instance's availability zone is not **1A** and **1B** then get the Subnet ID of the instance subnet then add it in the Load balancer, in my case it show subnet **1D**

```
SUB_D=$(aws ec2 describe-subnets --region us-east-1 --filters
"Name=vpc-id,Values=$VPC_ID" "Name=availability-zone,Values=us-east-1d"
--query 'Subnets[0].SubnetId' --output text)
```

**Create Security Group (xfusion-sg)**
This group will allow public web traffic **(Port 80)** to hit the Load Balancer

Create the Security Group

```
ALB_SG_ID=$(aws ec2 create-security-group --group-name xfusion-sg
--description "Security group for xfusion-alb" --vpc-id $VPC_ID --region
us-east-1 --query 'GroupId' --output text)

echo $ALB_SG_ID
```

Allow inbound HTTP traffic on port 80 from the world

```
aws ec2 authorize-security-group-ingress --group-id $ALB_SG_ID --protocol
tcp --port 80 --cidr 0.0.0.0/0 --region us-east-1
```

**Create Target Group and Register Instance**
The Target Group (`xfusion-tg`) tells the ALB where to send the traffic.
Create the Target Group

```
TG_ARN=$(aws elbv2 create-target-group --name xfusion-tg --protocol HTTP
--port 80 --vpc-id $VPC_ID --target-type instance --region us-east-1
--query 'TargetGroups[0].TargetGroupArn' --output text)

echo $TG_ARN
```

Register the xfusion-ec2 instance to this group

```
aws elbv2 register-targets --target-group-arn $TG_ARN --targets
Id=$INSTANCE_ID --region us-east-1
```

**Create the ALB (`xfusion-alb`)**
Create the Load Balancer

```
ALB_ARN=$(aws elbv2 create-load-balancer --name xfusion-alb --subnets
$SUB_A $SUB_B $SUB_D --security-groups $ALB_SG_ID --scheme internet-facing
--type application --region us-east-1 --query
'LoadBalancers[0].LoadBalancerArn' --output text)

echo $ALB_ARN
```

Create the Listener to forward Port 80 traffic to the Target Group

```
aws elbv2 create-listener --load-balancer-arn $ALB_ARN --protocol HTTP
--port 80 --default-actions Type=forward,TargetGroupArn=$TG_ARN --region
us-east-1
```

**Secure the EC2 Instance**

For security, EC2 instance should **only** accept traffic from the ALB, not the entire internet. We will update the EC2 instance's default security group to allow Port 80 specifically from `xfusion-sg`.

Get the current Security Group of the EC2 instance

```
EC2_SG_ID=$(aws ec2 describe-instances --instance-ids $INSTANCE_ID --query
'Reservations[0].Instances[0].SecurityGroups[0].GroupId' --output text)

echo $EC2_SG_ID
```

Allow inbound traffic from the ALB's Security Group only

```
aws ec2 authorize-security-group-ingress --group-id $EC2_SG_ID --protocol
tcp --port 80 --source-group $ALB_SG_ID --region us-east-1
```

**Verify**

Find the DNS name of the new ALB. Paste this into the browser to see Nginx sample page.

```
aws elbv2 describe-load-balancers --name xfusion-alb --region us-east-1
--query 'LoadBalancers[0].DNSName' --output text
```

# Day 25: Setting Up an EC2 Instance and CloudWatch Alarm

## Task Description:

The Nautilus DevOps team has been tasked with setting up an EC2 instance for their application. To ensure the application performs optimally, they also need to create a CloudWatch alarm to monitor the instance's CPU utilization. The alarm should trigger if the CPU utilization exceeds 90% for one consecutive 5-minute period. To send notifications, use the SNS topic named `nautilus-sns-topic` which is already created.

1. **Launch EC2 Instance:** Create an EC2 instance named `nautilus-ec2` using any appropriate Ubuntu AMI.
2. **Create CloudWatch Alarm:** Create a CloudWatch alarm named `nautilus-alarm` with the following specifications:
    - Statistic: Average
    - Metric: CPU Utilization
    - Threshold: >= 90% for 1 consecutive 5-minute period.
    - Alarm Actions: Send a notification to `nautilus-sns-topic`

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create key pair**

```
aws ec2 create-key-pair --region us-east-1 --key-name nautilus-kp
--key-type rsa --query 'KeyMaterial' --output text > nautilus-kp.pem
```

**To verify the key pair was created**

```
aws ec2 describe-key-pairs --region us-east-1 --key-name nautilus-kp
```

**Change Key Permission**

```
chmod 600 nautilus-kp.pem
```

**Retrieve the Latest AMI ID using SSM Parameter Store**

```
UBUNTU_AMI_ID=$(aws ssm get-parameter --region us-east-1 --name
/aws/service/canonical/ubuntu/server/noble/stable/current/amd64/hvm/ebs-gp3
/ami-id --query 'Parameter.Value' --output text)


echo "Using AMI ID: $UBUNTU_AMI_ID"
```

**Launch the EC2 Instance**

```
INSTANCE_ID=$(aws ec2 run-instances --region us-east-1 --image-id
$UBUNTU_AMI_ID --instance-type t2.micro --key-name nautilus-kp --count 1
--tag-specifications
'ResourceType=instance,Tags=[{Key=Name,Value=nautilus-ec2}]' --query
'Instances[0].InstanceId' --output text)

echo "Launched Instance ID: $INSTANCE_ID"
```

Since we did not pass the Subnet ID and Security group ID in the EC2 creation command, the reason for this was, If we did not mention the subnet ID and Security group ID it would pick the default VPC and Security group in that Region
**AWS Default VPC/Subnet Rule**

**Verification**

```
aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=nautilus-ec2" --query "Reservations[].Instances[].{
ID:InstanceId, State:State.Name, Type:InstanceType, KeyName:KeyName,
PublicIP:PublicIpAddress, VPC:VpcId, Subnet:SubnetId,
SecurityGroups:join(', ', NetworkInterfaces[].Groups[].GroupId) }" --output
```

```
table
```

**Get the SNS Topic ARN**

The alarm needs the full Amazon Resource Name (ARN) of the existing `nautilus-sns-topic` to send notifications.

```
SNS_ARN=$(aws sns list-topics --region us-east-1 --query
"Topics[?contains(TopicArn, 'nautilus-sns-topic')].TopicArn" --output text)

echo "SNS Topic ARN: $SNS_ARN"
```

Create the CloudWatch Alarm

```
aws cloudwatch put-metric-alarm --region us-east-1 --alarm-name
nautilus-alarm --alarm-description "Alarm when CPU exceeds 90% for 5
minutes" --metric-name CPUUtilization --namespace AWS/EC2 --statistic
Average --period 300 --threshold 90 --comparison-operator
GreaterThanOrEqualToThreshold --dimensions
Name=InstanceId,Value=$INSTANCE_ID --evaluation-periods 1 --alarm-actions
$SNS_ARN
```

**Verify**

```
aws cloudwatch describe-alarms --region us-east-1 --alarm-names
nautilus-alarm --query
'MetricAlarms[0].{Name:AlarmName,State:StateValue,Threshold:Threshold}'
```

# Day 26: Configuring an EC2 Instance as a Web Server with Nginx

## Task Description:

The Nautilus DevOps Team is working on setting up a new web server for a critical application. The team lead has requested you to create an EC2 instance that will serve as a web server using Nginx. This instance will be part of the initial infrastructure setup for the Nautilus project. Ensuring that the server is correctly configured and accessible from the internet is crucial for the upcoming deployment phase.

As a member of the Nautilus DevOps Team, your task is to create an EC2 instance with the following specifications:

**Instance Name:** The EC2 instance must be named `nautilus-ec2`.

**AMI:** Use any available Ubuntu AMI to create this instance.

**User Data Script:** Configure the instance to run a user data script during its launch. This script should:

- Install the Nginx package.
- Start the Nginx service.

**Security Group:** Ensure that the instance allows HTTP traffic on port `80` from the internet.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create the Web Server Security Group**

Create the Security Group

```
SG_ID=$(aws ec2 create-security-group --group-name nautilus-web-sg
--description "Security group for Nautilus web server" --region us-east-1
--query 'GroupId' --output text)
```

Allow Inbound HTTP (Port 80) from the internet

```
aws ec2 authorize-security-group-ingress --group-id $SG_ID --protocol tcp
--port 80 --cidr 0.0.0.0/0 --region us-east-1
```

**Create key pair**

```
aws ec2 create-key-pair --region us-east-1 --key-name nautilus-kp
--key-type rsa --query 'KeyMaterial' --output text > nautilus-kp.pem
```

**To verify the key pair was created**

```
aws ec2 describe-key-pairs --region us-east-1 --key-name nautilus-kp
```

**Change Key Permission**

```
chmod 600 nautilus-kp.pem
```

**Retrieve the Latest AMI ID using SSM Parameter Store**

```
UBUNTU_AMI_ID=$(aws ssm get-parameter --region us-east-1 --name
/aws/service/canonical/ubuntu/server/noble/stable/current/amd64/hvm/ebs-gp3
/ami-id --query 'Parameter.Value' --output text)


echo "Using AMI ID: $UBUNTU_AMI_ID"
```

**Prepare the Configuration Script (User Data)**

Create the script file

```
touch userdata.sh
chmod +x userdata.sh
```

Add the following content in userdata.sh file

```
#!/bin/bash

# Update package list and install Nginx
apt-get update -y
apt-get install nginx -y

# Ensure Nginx starts and is enabled on boot
systemctl start nginx
systemctl enable nginx
```

**Launch the EC2 Instance**

```
INSTANCE_ID=$(aws ec2 run-instances --region us-east-1 --image-id
$UBUNTU_AMI_ID --instance-type t2.micro --key-name nautilus-kp
--security-group-ids $SG_ID --user-data file://userdata.sh --count 1
--tag-specifications
'ResourceType=instance,Tags=[{Key=Name,Value=nautilus-ec2}]' --query
'Instances[0].InstanceId' --output text)

echo "Launched Instance ID: $INSTANCE_ID"
```

Since we did not pass the Subnet ID in the EC2 creation command, the reason for this was, If we did not mention the subnet ID it would pick the default VPC and Security group in that Region

**AWS Default VPC/Subnet Rule**

**Verification**

```
aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=nautilus-ec2" --query "Reservations[].Instances[].{
ID:InstanceId, State:State.Name, Type:InstanceType, KeyName:KeyName,
PublicIP:PublicIpAddress, VPC:VpcId, Subnet:SubnetId,
SecurityGroups:join(', ', NetworkInterfaces[].Groups[].GroupId) }" --output
table
```

# Day 27: Configuring a Public VPC with an EC2 Instance for Internet Access

## Task Description:

The Nautilus DevOps Team has received a request from the Networking Team to set up a new public VPC to support a set of public-facing services. This VPC will host various resources that need to be accessible over the internet. As part of this setup, you need to ensure the VPC has public subnets with automatic IP assignment for resources. Additionally, a new EC2 instance will be launched within this VPC to host public applications that require SSH access. This setup will enable the Networking Team to deploy and manage public-facing applications.

Create a public VPC named `xfusion-pub-vpc`, and a subnet named `xfusion-pub-subnet` under the same, make sure public IP is being auto assigned to resources under this subnet.

Create an EC2 instance named `xfusion-pub-ec2` under this VPC with instance type `t2.micro`. Make sure SSH port `22` is open for this instance and accessible over the internet.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)
***Notes:***
Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create the VPC and Subnet**

Create the VPC (xfusion-pub-vpc)

```
VPC_ID=$(aws ec2 create-vpc --region us-east-1 --cidr-block 10.0.0.0/16
--query 'Vpc.VpcId' --output text)

aws ec2 create-tags --region us-east-1 --resources $VPC_ID --tags
Key=Name,Value=xfusion-pub-vpc
```

Create the Subnet (xfusion-pub-subnet)

```
SUBNET_ID=$(aws ec2 create-subnet --region us-east-1 --vpc-id $VPC_ID
--cidr-block 10.0.1.0/24 --query 'Subnet.SubnetId' --output text)

aws ec2 create-tags --region us-east-1 --resources $SUBNET_ID --tags
Key=Name,Value=datacenter-pub-subnet
```

Enable Auto-assign Public IP

```
aws ec2 modify-subnet-attribute --region us-east-1 --subnet-id $SUBNET_ID
--map-public-ip-on-launch
```

**Configure Internet Connectivity**
For a subnet to be "public," it must have a route to an Internet Gateway (IGW).

Create and Attach an Internet Gateway on VPC

```
IGW_ID=$(aws ec2 create-internet-gateway --region us-east-1 --query
'InternetGateway.InternetGatewayId' --output text)

aws ec2 attach-internet-gateway --region us-east-1 --internet-gateway-id
$IGW_ID --vpc-id $VPC_ID
```

Create a Route Table and add a route to the Internet

```
RT_ID=$(aws ec2 create-route-table --region us-east-1 --vpc-id $VPC_ID
--query 'RouteTable.RouteTableId' --output text)

aws ec2 create-route --region us-east-1 --route-table-id $RT_ID
--destination-cidr-block 0.0.0.0/0 --gateway-id $IGW_ID
```

Associate the Route Table with Subnet

```
aws ec2 associate-route-table --region us-east-1 --route-table-id $RT_ID
```

```
--subnet-id $SUBNET_ID
```

**Create Security Group**

```
SG_ID=$(aws ec2 create-security-group --region us-east-1 --group-name
xfusion-ssh-sg --description "Allow SSH access" --vpc-id $VPC_ID --query
'GroupId' --output text)
```

Open Port 22 for SSH from anywhere

```
aws ec2 authorize-security-group-ingress --region us-east-1 --group-id
$SG_ID --protocol tcp --port 22 --cidr 0.0.0.0/0
```

**Retrieve the Latest AMI ID using SSM Parameter Store**

```
UBUNTU_AMI_ID=$(aws ssm get-parameter --region us-east-1 --name
/aws/service/canonical/ubuntu/server/noble/stable/current/amd64/hvm/ebs-gp3
/ami-id --query 'Parameter.Value' --output text)


echo "Using AMI ID: $UBUNTU_AMI_ID"
```

**Create key pair**

```
aws ec2 create-key-pair --region us-east-1 --key-name xfusion-kp --key-type
rsa --query 'KeyMaterial' --output text > nautilus-kp.pem
```

**To verify the key pair was created**

```
aws ec2 describe-key-pairs --region us-east-1 --key-name xfusion-kp
```

**Change Key Permission**

```
chmod 600 xfusion-kp.pem
```

**Launch EC2 Instance** (xfusion-pub-ec2)

```
aws ec2 run-instances --region us-east-1 --image-id $UBUNTU_AMI_ID --count
1 --instance-type t2.micro --key-name xfusion-kp --subnet-id $SUBNET_ID
--security-group-ids $SG_ID --tag-specifications
'ResourceType=instance,Tags=[{Key=Name,Value=xfusion-pub-ec2}]'
```

**Verification**

```
aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=xfusion-pub-ec2" --query
```

```
"Reservations[].Instances[].{ ID:InstanceId, State:State.Name,
Type:InstanceType, KeyName:KeyName, PublicIP:PublicIpAddress, VPC:VpcId,
Subnet:SubnetId, SecurityGroups:join(', ',
NetworkInterfaces[].Groups[].GroupId) }" --output table
```

# Day 28: Creating a Private ECR Repository

## Task Description:

The Nautilus DevOps team has been tasked with setting up a containerized application. They need to create a private Amazon Elastic Container Registry (ECR) repository to store their Docker images. Once the repository is created, they will build a Docker image from a Dockerfile located on the `aws-client` host and push this image to the ECR repository. This process is essential for maintaining and deploying containerized applications in a streamlined manner.

Create a private ECR repository named `datacenter-ecr`. There is a Dockerfile under `/root/pyapp` directory on `aws-client` host, build a docker image using this Dockerfile and push the same to the newly created ECR repo, the image tag must be `latest`.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)
***Notes:***
Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create the Private ECR Repository**

```
aws ecr create-repository --repository-name datacenter-ecr --region
us-east-1 --query 'repository.repositoryUri' --output text
```

**Note:** Save the output URI (e.g.,

`123456789012.dkr.ecr.us-east-1.amazonaws.com/datacenter-ecr`), as we will need it for tagging.

**ECR Authentication**

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin $(aws sts get-caller-identity --query 'Account' --output text).dkr.ecr.us-east-1.amazonaws.com
```

```
~ on ☁️  (us-east-1) ➜  docker --version
Docker version 20.10.5+dfsg1, build 55c4c88

~ on ☁️  (us-east-1) ➜  aws ecr get-login-password --region us-east-1 | docker login --username AWS --password
-stdin $(aws sts get-caller-identity --query 'Account' --output text).dkr.ecr.us-east-1.amazonaws.com
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded

~ on ☁️  (us-east-1) ➜
```

**Build and Tag the Image**

Change directory to where the Dockerfile is located

```
cd /root/pyapp
```

Build the image locally

```
docker build -t datacenter-ecr:latest .
```

```
~/pyapp on ☁️  (us-east-1) ➜  docker images
REPOSITORY       TAG        IMAGE ID       CREATED              SIZE
datacenter-ecr   latest     c297213639c6   About a minute ago   131MB
python           3.8-slim   b5f62925bd0f   16 months ago        125MB
```

Set the ECR URI

```
ECR_URI="<ECR_URI>"
```

Tag the image for ECR repository

```
docker tag datacenter-ecr:latest $ECR_URI:latest
```

```
~/pyapp on ☁️  (us-east-1) ➜  docker tag datacenter-ecr:latest $ECR_URI:latest

~/pyapp on ☁️  (us-east-1) ➜  docker images
REPOSITORY                                                        TAG       IMAGE ID       CREATED          SIZE
453274592256.dkr.ecr.us-east-1.amazonaws.com/datacenter-ecr       latest    c297213639c6   3 minutes ago    131MB
datacenter-ecr                                                    latest    c297213639c6   3 minutes ago    131MB
python                                                            3.8-slim  b5f62925bd0f   16 months ago    125MB
```

### Push the Image to ECR

```
docker push $ECR_URI:latest
```

### Verify

```
aws ecr list-images --repository-name datacenter-ecr --region us-east-1
```

# Day 29: Establishing Secure Communication Between Public and Private VPCs via VPC Peering

## Task Description:

The Nautilus DevOps team has been tasked with demonstrating the use of VPC Peering to enable communication between two VPCs. One VPC will be a private VPC that contains a private EC2 instance, while the other will be the default public VPC containing a publicly accessible EC2 instance.

1) There is already an existing **EC2 instance** in the public vpc/subnet:
   - Name: `datacenter-public-ec2`

2) There is already an existing **Private VPC**:
   - Name: `datacenter-private-vpc`
   - CIDR: `10.1.0.0/16`

3) There is already an existing **Subnet** in `datacenter-private-vpc`:
   - Name: `datacenter-private-subnet`
   - CIDR: `10.1.1.0/24`

4) There is already an existing **EC2 instance** in the private subnet:
   - Name: `datacenter-private-ec2`

5) **Create a Peering Connection** between the Default VPC and the Private VPC:
   - VPC Peering Connection Name: `datacenter-vpc-peering`

6) **Configure Route Tables** to enable communication between the two VPCs.
   - Ensure the private EC2 instance is accessible from the public EC2 instance.

7) **Test the Connection**:
   - Add `/root/.ssh/id_rsa.pub` public key to the public EC2 instance's `ec2-user`'s `authorized_keys` to make sure we are able to ssh into this instance from AWS client host. You may also need to update the security group of the private EC2 instance to allow ICMP traffic from the public/default VPC CIDR. This will enable you to ping the private instance from the public instance.
   - SSH into the public EC2 instance and ensure that you can ping the private EC2 instance.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)
***Notes:*** Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Establish VPC Peering**

Get the VPC IDs

```
PUBLIC_VPC_ID=$(aws ec2 describe-vpcs --region us-east-1 --filters
"Name=isDefault,Values=true" --query 'Vpcs[0].VpcId' --output text)

echo $PUBLIC_VPC_ID


PRIVATE_VPC_ID=$(aws ec2 describe-vpcs --region us-east-1 --filters
"Name=tag:Name,Values=datacenter-private-vpc" --query 'Vpcs[0].VpcId'
--output text)

echo $PRIVATE_VPC_ID
```

Request Peering Connection

```
PEERING_ID=$(aws ec2 create-vpc-peering-connection --region us-east-1
--vpc-id $PUBLIC_VPC_ID --peer-vpc-id $PRIVATE_VPC_ID --tag-specifications
'ResourceType=vpc-peering-connection,Tags=[{Key=Name,Value=datacenter-vpc-p
eering}]' --query 'VpcPeeringConnection.VpcPeeringConnectionId' --output
text)

echo $PEERING_ID
```

Accept the Peering Connection

```
aws ec2 accept-vpc-peering-connection --region us-east-1
--vpc-peering-connection-id $PEERING_ID
```

**Configure Route Tables**

For traffic to flow, each VPC needs a route pointing to the other VPC's CIDR via the Peering Connection.

Get Route Table IDs

```
PUBLIC_RT=$(aws ec2 describe-route-tables --region us-east-1 --filters
"Name=vpc-id,Values=$PUBLIC_VPC_ID" --query 'RouteTables[0].RouteTableId'
--output text)

echo $PUBLIC_RT

PRIVATE_RT=$(aws ec2 describe-route-tables --region us-east-1 --filters
"Name=vpc-id,Values=$PRIVATE_VPC_ID" --query 'RouteTables[0].RouteTableId'
--output text)

echo $PRIVATE_RT
```

Add route in Public VPC to Private VPC (10.1.0.0/16)

```
aws ec2 create-route --region us-east-1 --route-table-id $PUBLIC_RT
--destination-cidr-block 10.1.0.0/16 --vpc-peering-connection-id
$PEERING_ID
```

Add route in Private VPC to Public VPC (Usually 172.31.0.0/16 for Default VPC)

```
DEFAULT_CIDR=$(aws ec2 describe-vpcs --vpc-ids $PUBLIC_VPC_ID --region
us-east-1 --query 'Vpcs[0].CidrBlock' --output text)

echo $DEFAULT_CIDR

aws ec2 create-route --region us-east-1 --route-table-id $PRIVATE_RT
--destination-cidr-block $DEFAULT_CIDR --vpc-peering-connection-id
$PEERING_ID
```

**Update Security Groups (Allow ICMP)**

Get the Security Group ID of the private instance

```
PRIVATE_SG=$(aws ec2 describe-instances --region us-east-1 --filters
```

```
"Name=tag:Name,Values=datacenter-private-ec2" --query
'Reservations[0].Instances[0].SecurityGroups[0].GroupId' --output text)

echo $PRIVATE_SG
```

Allow ICMP (Ping) from the Default VPC CIDR

```
aws ec2 authorize-security-group-ingress --region us-east-1 --group-id
$PRIVATE_SG --protocol icmp --port -1 --cidr $DEFAULT_CIDR
```

**Configure SSH Access**

Get the Public IP of the public instance

```
PUBLIC_IP=$(aws ec2 describe-instances --region us-east-1 --filters
"Name=tag:Name,Values=datacenter-public-ec2" --query
'Reservations[0].Instances[0].PublicIpAddress' --output text)

echo $PUBLIC_IP
```

Append the public key to the ec2-user's authorized_keys

```
cat /root/.ssh/id_rsa.pub | ssh ec2-user@$PUBLIC_IP "cat >>
~/.ssh/authorized_keys"
```

**Test the Connection**
**SSH into the Public Instance:** `ssh ec2-user@$PUBLIC_IP`
**Ping the Private Instance:** From the public instance shell, ping the private IP of `datacenter-private-ec2`.

# Day 30: Enable Internet Access for Private EC2 using NAT Instance

## Task Description:

The Nautilus DevOps team is tasked with enabling internet access for an EC2 instance running in a private subnet. This instance should be able to upload a test file to a public S3 bucket once it can access the internet. To minimize costs, the team has decided to use a NAT Instance instead of a NAT Gateway.

The following components already exist in the environment:

1. A VPC named `devops-priv-vpc` and a private subnet named `devops-priv-subnet` have been created.
2. An EC2 instance named `devops-priv-ec2` is already running in the private subnet.
3. The EC2 instance is configured with a cron job that uploads a test file to the S3 bucket `devops-nat-13120` every minute. Upload will only succeed once internet access is established.

Your task is to:

- Create a new public subnet named `devops-pub-subnet` in the existing VPC.
- Launch a NAT Instance in the public subnet using an Amazon Linux 2 AMI and name it `devops-nat-instance`. Configure this instance to act as a NAT instance. Make sure to use a custom security group for this instance.

After the configuration, verify that the test file `devops-test.txt` appears in the S3 bucket `devops-nat-13120`. This indicates successful internet access from the private EC2 instance via the NAT Instance.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

**NAT Instance**

A **NAT Instance** is a self-managed EC2 instance that acts as a bridge, allowing private servers in your VPC to reach the internet (for updates or API calls) while preventing the internet from starting connections with them.

While AWS recommends **NAT Gateways** for production because they are managed and highly available, **NAT Instances** remain popular for development and lab environments because they are significantly cheaper (you only pay for the EC2 runtime).

**How a NAT Instance Works**

The NAT Instance sits in a **Public Subnet** and has a public IP. When a private instance wants to reach the internet, it sends its traffic to the NAT Instance. The NAT Instance then "masquerades" that traffic as its own before sending it to the Internet Gateway.

To make this work, two things must happen at the **AWS Infrastructure level**:

1. **Route Table:** The private subnet's route table must have an entry: `0.0.0.0/0 -> [NAT Instance ID]`.
2. **Source/Destination Check:** By default, EC2 instances drop any packet that isn't addressed to them. Since a NAT Instance is a "middleman" receiving traffic meant for the internet, you must **Disable Source/Destination Check** in the EC2 Console (`Actions > Networking > Change source/destination check`).

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Get the Private VPC ID**

```
VPC_ID=$(aws ec2 describe-vpcs --region us-east-1 --filters
"Name=tag:Name,Values=devops-priv-vpc" --query 'Vpcs[0].VpcId' --output
text)

echo $VPC_ID
```

**Create the Public Subnet (devops-pub-subnet)**

Get the CIDR of Private VPC

```
PRIVATE_VPC_CIDR=$(aws ec2 describe-vpcs --vpc-ids $VPC_ID --region
us-east-1 --query 'Vpcs[0].CidrBlock' --output text)

echo $PRIVATE_VPC_CIDR
```

If the CIDR range is 10.1.0.0/16 then use 10.1.2.0/24 in the next command

```
PUB_SUBID=$(aws ec2 create-subnet --region us-east-1 --vpc-id $VPC_ID
--cidr-block 10.1.2.0/24 --tag-specifications
'ResourceType=subnet,Tags=[{Key=Name,Value=devops-pub-subnet}]' --query
'Subnet.SubnetId' --output text)

echo $PUB_SUBID
```

Associate with the Main Route Table (which should have the IGW)

```
RT_PUB=$(aws ec2 describe-route-tables --region us-east-1 --filters
"Name=vpc-id,Values=$VPC_ID" "Name=tag:Name,Values=devops-pub-rt" --query
'RouteTables[0].RouteTableId' --output text)

echo $RT_PUB

aws ec2 associate-route-table --region us-east-1 --subnet-id $PUB_SUBID
--route-table-id $RT_PUB
```

**Configure Internet Connectivity**
For a subnet to be "public," it must have a route to an Internet Gateway (IGW).

Create and Attach an Internet Gateway on VPC

```
IGW_ID=$(aws ec2 create-internet-gateway --region us-east-1 --query
'InternetGateway.InternetGatewayId' --output text)

aws ec2 attach-internet-gateway --region us-east-1 --internet-gateway-id
$IGW_ID --vpc-id $VPC_ID
```

Add a route to the Internet in the devops-pub-rt

```
aws ec2 create-route --region us-east-1 --route-table-id $RT_PUB
--destination-cidr-block 0.0.0.0/0 --gateway-id $IGW_ID
```

Associate the Route Table with Subnet

```
aws ec2 associate-route-table --region us-east-1 --route-table-id $RT_PUB
--subnet-id $PUB_SUBID
```

**Launch the NAT Instance**

**Create a Custom Security Group for NAT**

```
NAT_SG=$(aws ec2 create-security-group --region us-east-1 --group-name
devops-nat-sg --description "NAT Instance SG" --vpc-id $VPC_ID --query
'GroupId' --output text)

echo $NAT_SG
```

Allow Inbound Traffic from the Private Subnet (Port 80/443 or All for testing)

```
aws ec2 authorize-security-group-ingress --region us-east-1 --group-id
$NAT_SG --protocol all --port -1 --cidr 10.1.1.0/24
```

**Create Key pair**

```
aws ec2 create-key-pair --region us-east-1 --key-name devops-nat-kp
--key-type rsa --query 'KeyMaterial' --output text > devops-nat-kp.pem
```

**To verify the key pair was created**

```
aws ec2 describe-key-pairs --region us-east-1 --key-name devops-nat-kp
```

**Change Key Permission**

```
chmod 600 devops-nat-kp.pem
```

**Retrieve the Latest AMI ID using SSM Parameter Store**

```
AMI_ID=$(aws ssm get-parameter --region us-east-1 --name
/aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-x86_64-gp2 --query
'Parameter.Value' --output text)

echo "Using AMI ID: $AMI_ID"
```

**Prepare the Configuration Script (User Data)**

Create the script file

```
touch userdata.sh
chmod +x userdata.sh
```

Add the following content in userdata.sh file

```
#!/bin/bash

sudo yum install iptables-services -y

# Install and start the firewall service
sudo systemctl enable iptables
sudo systemctl start iptables

# Enable IP Forwarding in the Linux Kernel
# This tells the OS it is allowed to pass packets between interfaces.
sudo echo "net.ipv4.ip_forward=1" >> /etc/sysctl.conf
sudo sysctl -p

# Configure the NAT Masquerade rule
# -t nat: Use the Network Address Translation table.
# -A POSTROUTING: Apply this rule to packets leaving the instance.
# -o eth0: Use the primary network interface.
# -j MASQUERADE: Replace the private source IP with the NAT instance's IP.
sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE

# Save the rules so they persist after a reboot
sudo service iptables save
```

**Launch the NAT Instance**

```
NAT_INST_ID=$(aws ec2 run-instances --region us-east-1 --image-id $AMI_ID
--instance-type t2.micro --key-name devops-nat-kp --subnet-id $PUB_SUBID
--security-group-ids $NAT_SG --associate-public-ip-address \
    --user-data file://userdata.sh --tag-specifications
'ResourceType=instance,Tags=[{Key=Name,Value=devops-nat-instance}]' --query
'Instances[0].InstanceId' --output text)
```

CRITICAL: Disable Source/Destination Checks

```
aws ec2 modify-instance-attribute --region us-east-1 --instance-id
$NAT_INST_ID --no-source-dest-check
```

**Update Private Subnet Route Table**

Now, tell the private subnet that the "gateway" to the internet is the NAT Instance.

Get the Private Route Table

```
PRIV_RT=$(aws ec2 describe-route-tables --region us-east-1 --filters
"Name=association.subnet-id,Values=$(aws ec2 describe-subnets --filters
"Name=tag:Name,Values=devops-priv-subnet" --query 'Subnets[0].SubnetId'
--output text)" --query 'RouteTables[0].RouteTableId' --output text)

echo $PRIV_RT
```

Add Default Route (0.0.0.0/0) pointing to the NAT Instance

```
aws ec2 create-route --region us-east-1 --route-table-id $PRIV_RT
--destination-cidr-block 0.0.0.0/0 --instance-id $NAT_INST_ID
```

**Verification**
After waiting ~2 minutes for the NAT instance to initialize:

1. **Check S3:** Run `aws s3 ls s3://devops-nat-13120` to see if `devops-test.txt` has appeared.
2. **Check Logs:** If it hasn't appeared, check the NAT instance logs to ensure `iptables` and `ip_forwarding` are active.

# Day 31: Configuring a Private RDS Instance for Application Development

## Task Description:

The Nautilus Development Team is working on a new application feature that requires a reliable and scalable database solution. To facilitate development and testing, they need a new private RDS instance. This instance will be used to store critical application data and must be provisioned using the AWS free tier to minimize costs during the initial development phase. The team has chosen **MySQL** as the database engine due to its compatibility with their existing systems. The DevOps team has been tasked with setting up this RDS instance, ensuring that it is correctly configured and available for use by the development team.

As a member of the Nautilus DevOps Team, your task is to perform the following:

1. **Provision a Private RDS Instance:** Create a new private RDS instance named `xfusion-rds` using a `sandbox` template, further it must be a `db.t3.micro` type instance.
2. **Engine Configuration:** Use the `MySQL` engine with version `8.4.x`.
3. **Enable Storage Autoscaling:** Enable storage autoscaling and set the threshold value to `50GB`. Keep the rest of the configurations as default.
4. **Instance Availability:** Ensure the instance is in the `available` state before submitting this task.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Provision the RDS Instance**

```
aws rds create-db-instance --region us-east-1 --db-instance-identifier
xfusion-rds --db-instance-class db.t3.micro --engine mysql --engine-version
8.4 --allocated-storage 20 --max-allocated-storage 50 --master-username
admin --master-user-password "<PASSWORD>" --no-publicly-accessible
--storage-type gp3 --tags Key=Name,Value=xfusion-rds
```

Change the `MASTER_PASSWORD` to a secure value of choice before running the command.

**Verification**

```
aws rds describe-db-instances --db-instance-identifier xfusion-rds --region
us-east-1 --query
'DBInstances[0].{Status:DBInstanceStatus,Engine:Engine,Version:EngineVersio
n,MaxStorage:MaxAllocatedStorage,Public:PubliclyAccessible}'
```

# Day 32: Snapshot and Restoration of an RDS Instance

## Task Description:

The Nautilus Development Team is preparing for a major update to their database infrastructure. To ensure a smooth transition and to safeguard data, the team has requested the DevOps team to take a snapshot of the current RDS instance and restore it to a new instance. This process is crucial for testing and validation purposes before the update is rolled out to the production environment. The snapshot will serve as a backup, and the new instance will be used to verify that the backup process works correctly and that the application can function seamlessly with the restored data.

As a member of the Nautilus DevOps Team, your task is to perform the following:

1. **Take a Snapshot:** Take a snapshot of the `devops-rds` RDS instance and name it `devops-snapshot` (please wait `devops-rds` instance to be in `available` state).
2. **Restore the Snapshot:** Restore the snapshot to a new RDS instance named `devops-snapshot-restore`.
3. **Configure the New RDS Instance:** Ensure that the new RDS instance has a class of `db.t3.micro`.
4. **Verify the New RDS Instance:** The new RDS instance must be in the `Available` state upon completion of the restoration process.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Verify Source Instance State**

```
aws rds wait db-instance-available --db-instance-identifier devops-rds
--region us-east-1
```

**Take a Snapshot**

```
aws rds create-db-snapshot --db-instance-identifier devops-rds
--db-snapshot-identifier devops-snapshot --region us-east-1

# Wait for the snapshot to reach the available state
aws rds wait db-snapshot-available --db-snapshot-identifier devops-snapshot
--region us-east-1
```

**Restore the Snapshot to a New Instance**

```
aws rds restore-db-instance-from-db-snapshot --db-instance-identifier
devops-snapshot-restore --db-snapshot-identifier devops-snapshot
--db-instance-class db.t3.micro --region us-east-1
```

**Verify Restoration**

Wait for the restored instance to be available

```
aws rds wait db-instance-available --db-instance-identifier
devops-snapshot-restore --region us-east-1
```

Final verification check

```
aws rds describe-db-instances --db-instance-identifier
devops-snapshot-restore --region us-east-1 --query
'DBInstances[0].{Identifier:DBInstanceIdentifier,Status:DBInstanceStatus,Cl
ass:DBInstanceClass}'
```

# Day 33: Create a Lambda Function

## Task Description:

The Nautilus DevOps team is embracing serverless architecture by integrating AWS Lambda into their operational tasks. They have decided to deploy a simple Lambda function that will return a custom greeting to demonstrate serverless capabilities effectively. This function is crucial for showcasing rapid deployment and easy scalability features of AWS Lambda to the team.

1. **Create Lambda Function:** Create a Lambda function named `datacenter-lambda`.
2. **Runtime:** Use the Runtime `Python`.
3. **Deploy:** The function should print the body `Welcome to KKE AWS Labs!`.
4. **Status Code:** Ensure the status code is `200`.
5. **IAM Role:** Create and use the IAM role named `lambda_execution_role`.

Use the AWS Console to complete this task

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)
***Notes:***
Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

**Create the IAM Execution Role**
Before creating the function, we must define the permissions it will use.

1. Navigate to the **IAM Console**.
2. Click **Roles** > **Create role**.
3. **Trusted entity type:** Select **AWS service**.
4. **Service or use case:** Select **Lambda**. Click **Next**.
5. **Add permissions:** Search for and select the managed policy `AWSLambdaBasicExecutionRole`. This allows the function to upload logs to CloudWatch. Click **Next**.
6. **Role name:** Enter `lambda_execution_role`
7. Click **Create role**.

**Create the Lambda Function**

Now, create the function in the `us-east-1` region.

1. Navigate to the **Lambda Console**.
2. Click **Create function**.
3. Select **Author from scratch**.
4. **Function name:** `datacenter-lambda`
5. **Runtime:** Select **Python 3.x** (e.g., Python 3.12).
6. **Permissions:** Expand "Change default execution role".
    - Select **Use an existing role**.
    - Choose the `lambda_execution_role` you created in Step 1.
7. Click **Create function**.

**Deploy the Greeting Code**

Once the function is created, replace the default code with the greeting logic.

1. In the **Code source** section, open `lambda_function.py`.
2. Replace the existing code with the following snippet:

```
import json

def lambda_handler(event, context):
    # Print the body as requested for logging/visibility
    message = "Welcome to KKE AWS Labs!"
    print(message)

    return {
        'statusCode': 200,
        'body': json.dumps(message)
    }
```

Click **Deploy** to save and activate the changes.

**Verification**

To ensure the function is working as intended:

1. Click the **Test** button.
2. Create a new test event (name it `TestGreeting`, keep defaults) and click **Save**.
3. Click **Test** again.
4. The **Execution result** should show:
    - **Status:** Succeeded
    - **Response body:** `"Welcome to KKE AWS Labs!"`
    - **Function Logs:** Should display `Welcome to KKE AWS Labs!`

# Day 34: Create a Lambda Function Using CLI

## Task Description:

The Nautilus DevOps team continues to explore serverless architecture by setting up another Lambda function. The function will return a custom greeting and demonstrate the capabilities of AWS Lambda effectively.

1. **Create Python Script:** Create a Python script named `lambda_function.py` with a function that returns the body `Welcome to KKE AWS Labs!` and status code `200`.
2. **Zip the Python Script:** Zip the script into a file named `function.zip`.
3. **Create Lambda Function:** Create a Lambda function named `devops-lambda-cli` using the zipped file and specify `Python` as the runtime.
4. **IAM Role:** Use the IAM role named `lambda_execution_role`.

Use AWS CLI which is already configured on the `aws-client` host.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create the Python Script**

Create a file named `lambda_function.py` with the required greeting logic

```
import json

def lambda_handler(event, context):
    return {
        'statusCode': 200,
        'body': 'Welcome to KKE AWS Labs!'
    }
```

**Zip the Script**

Compress the Python file into a deployment package named `function.zip`.

```
zip function.zip lambda_function.py
```

**Get the Role ARN**

```
ROLE_ARN=$(aws iam get-role --role-name lambda_execution_role --query
'Role.Arn' --output text)

echo $ROLE_ARN
```

**Create the Lambda Function**

```
aws lambda create-function --region us-east-1 --function-name
devops-lambda-cli --runtime python3.12 --handler
lambda_function.lambda_handler --role $ROLE_ARN --zip-file
fileb://function.zip
```

**Verification**

Invoke the function

```
aws lambda invoke --function-name devops-lambda-cli --region us-east-1
response.json
```

View the response body

```
cat response.json
```

**Expected Result:** `{"statusCode": 200, "body": "Welcome to KKE AWS Labs!"}`

# Day 35: Deploying and Managing Applications on AWS

## Task Description:

The Nautilus DevOps team needs a new private RDS instance for their application. They need to set up a MySQL database and ensure that their existing EC2 instance can connect to it. This will help in managing their database needs efficiently and securely.

1) **Task Details:**

1. Create a private RDS instance named `nautilus-rds` using a `sandbox` template.
2. The engine type must be `MySQL v8.4.5`, and it must be a `db.t3.micro` type instance.
3. The master username must be `nautilus_admin` with an appropriate password.
4. The RDS storage type must be `gp2`, and the storage size must be `5GiB`.
5. Create a database named `nautilus_db`.
6. Keep the rest of the configurations as `default`. Ensure the instance is in `available` state.
7. Adjust the security groups so that the `nautilus-ec2` instance can connect to the RDS on `port 3306` and also open port `80` for the instance.

2) An EC2 instance named `nautilus-ec2` exists. Connect to this instance from the AWS console. Create an SSH key (`/root/.ssh/id_rsa`) on the `aws-client` host if it doesn't already exist. Add the public key to the authorized keys of the `root` user on the EC2 instance for password-less SSH access.

3) There is a file named `index.php` under the `/root` directory on the `aws-client` host. Copy this file to the `nautilus-ec2` instance under the `/var/www/html/` directory. Make the appropriate changes in the file to connect to the RDS.

4) You should see a `Connected successfully` message in the browser once you access the instance using the public IP.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Provision the Private RDS Instance**

```
aws rds create-db-instance --region us-east-1 --db-instance-identifier
nautilus-rds --db-instance-class db.t3.micro --engine mysql
--engine-version 8.4.5 --allocated-storage 5 --storage-type gp2 --db-name
nautilus_db --master-username nautilus_admin --master-user-password
"nautilus_password" --no-publicly-accessible
```

Note: Set the password in the above command.

**Configure Security Groups**

Get Security Group IDs

```
EC2_SG=$(aws ec2 describe-instances --filters
"Name=tag:Name,Values=nautilus-ec2" --query
'Reservations[0].Instances[0].SecurityGroups[0].GroupId' --output text)

echo $EC2_SG

RDS_SG=$(aws rds describe-db-instances --db-instance-identifier
nautilus-rds --query
'DBInstances[0].VpcSecurityGroups[0].VpcSecurityGroupId' --output text)

echo $RDS_SG
```

Allow port 3306 on RDS from the EC2 Security Group

```
aws ec2 authorize-security-group-ingress --group-id $RDS_SG --protocol tcp
```

```
--port 3306 --source-group $EC2_SG
```

Allow port 80 on EC2 from anywhere

```
aws ec2 authorize-security-group-ingress --group-id $EC2_SG --protocol tcp
--port 80 --cidr 0.0.0.0/0
```

**Set up Password-less SSH**

Create key if not exists

```
if [ ! -f /root/.ssh/id_rsa ]; then ssh-keygen -t rsa -N "" -f
/root/.ssh/id_rsa; fi
```

Get EC2 Public IP

```
EC2_IP=$(aws ec2 describe-instances --filters
"Name=tag:Name,Values=nautilus-ec2" --query
'Reservations[0].Instances[0].PublicIpAddress' --output text)

echo $EC2_IP
```

Add public key to EC2

```
PUB_KEY=$(cat /root/.ssh/id_rsa.pub)

ssh root@$EC2_IP "sudo mkdir -p /root/.ssh && echo '$PUB_KEY' | sudo tee -a
/root/.ssh/authorized_keys"
```

**Deploy and Configure PHP**

Get RDS Endpoint

```
RDS_ENDPOINT=$(aws rds describe-db-instances --db-instance-identifier
nautilus-rds --query 'DBInstances[0].Endpoint.Address' --output text)

echo $RDS_ENDPOINT
```

Update index.php with the Endpoint, Username, and Database

```
# Use sed to replace placeholders (assuming index.php has them)
sed -i "s/db_host_placeholder/$RDS_ENDPOINT/g" /root/index.php
sed -i "s/db_user_placeholder/nautilus_admin/g" /root/index.php
sed -i "s/db_pass_placeholder/nautilus_password/g" /root/index.php
```

```
sed -i "s/db_name_placeholder/nautilus_db/g" /root/index.php
```

Transfer file to EC2

```
scp /root/index.php root@$EC2_IP:/var/www/html/index.php
```

**Final Check**

Open browser and navigate to `http://<EC2_PUBLIC_IP>`. If everything is configured correctly, the PHP script will execute and display: **Connected successfully**

# Day 36: Load Balancing EC2 Instances with Application Load Balancer

## Task Description:

The Nautilus Development Team needs to set up a new EC2 instance and configure it to run a web server. This EC2 instance should be part of an Application Load Balancer (ALB) setup to ensure high availability and better traffic management. The task involves creating an EC2 instance, setting up an ALB, configuring a target group, and ensuring the web server is accessible via the ALB DNS.

**Create a security group:** Create a security group named `devops-sg` to open port `80` for the `default` security group (which will be attached to the ALB). Attach `devops-sg` security group to the EC2 instance.

**Create an EC2 instance:** Create an EC2 instance named `devops-ec2`. Use any available Ubuntu AMI to create this instance. Configure the instance to run a user data script during its launch.

This script should:

- Install the Nginx package.
- Start the Nginx service.

**Set up an Application Load Balancer:** Set up an Application Load Balancer named `devops-alb`. Attach `default` security group to the same.

**Create a target group:** Create a target group named `devops-tg`.

**Route traffic:** The ALB should route traffic on port `80` to port `80` of the `devops-ec2` instance.

**Security group adjustments:** Make appropriate changes in the `default` security group attached to the ALB if necessary. Eventually, the Nginx server running under `devops-ec2` instance must be accessible using the ALB DNS.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create Security Groups**
We need two security groups: one for the ALB (to accept web traffic) (default) and one for the EC2 instance (to accept traffic from the ALB).

Get the Default VPC ID

```
VPC_ID=$(aws ec2 describe-vpcs --filters "Name=isDefault,Values=true"
--query 'Vpcs[0].VpcId' --output text)
```

Create **devops-sg** for the EC2 instance

```
EC2_SG_ID=$(aws ec2 create-security-group --group-name devops-sg
--description "SG for devops-ec2" --vpc-id $VPC_ID --query 'GroupId'
--output text)
```

Get the Default Security Group ID (to be used for the ALB)

```
DEFAULT_SG_ID=$(aws ec2 describe-security-groups --filters
"Name=vpc-id,Values=$VPC_ID" "Name=group-name,Values=default" --query
'SecurityGroups[0].GroupId' --output text)
```

Add Rule: Allow Port 80 on Default SG (ALB) from anywhere

```
aws ec2 authorize-security-group-ingress --group-id $DEFAULT_SG_ID
--protocol tcp --port 80 --cidr 0.0.0.0/0
```

Add Rule: Allow Port 80 on devops-sg (EC2) only from the Default SG (ALB)

```
aws ec2 authorize-security-group-ingress --group-id $EC2_SG_ID --protocol
tcp --port 80 --source-group $DEFAULT_SG_ID
```

**Launch the EC2 Instance with Nginx**

**Create key pair**

```
aws ec2 create-key-pair --region us-east-1 --key-name devops-kp --key-type
rsa --query 'KeyMaterial' --output text > devops-kp.pem
```

**To verify the key pair was created**

```
aws ec2 describe-key-pairs --region us-east-1 --key-name devops-kp
```

**Change Key Permission**

```
chmod 600 devops-kp.pem
```

**Retrieve the Latest AMI ID using SSM Parameter Store**

```
UBUNTU_AMI_ID=$(aws ssm get-parameter --region us-east-1 --name
/aws/service/canonical/ubuntu/server/noble/stable/current/amd64/hvm/ebs-gp3
/ami-id --query 'Parameter.Value' --output text)


echo "Using AMI ID: $UBUNTU_AMI_ID"
```

**Prepare the Configuration Script (User Data)**

Create the script file

```
touch userdata.sh
chmod +x userdata.sh
```

Add the following content in userdata.sh file

```
#!/bin/bash

# Update package list and install Nginx
sudo apt-get update -y
sudo apt-get install nginx -y

# Ensure Nginx starts and is enabled on boot
sudo systemctl start nginx
sudo systemctl enable nginx
```

Launch Instance

```
INSTANCE_ID=$(aws ec2 run-instances --image-id $UBUNTU_AMI_ID --count 1
--instance-type t2.micro --security-group-ids $EC2_SG_ID --key-name
devops-kp --user-data file://userdata.sh --tag-specifications
'ResourceType=instance,Tags=[{Key=Name,Value=devops-ec2}]' --query
'Instances[0].InstanceId' --output text)

echo $INSTANCE_ID
```

**Configure Target Group and ALB**

Create the Target Group, register the instance, and set up the ALB with a listener.

Create Target Group

```
TG_ARN=$(aws elbv2 create-target-group --name devops-tg --protocol HTTP
--port 80 --vpc-id $VPC_ID --target-type instance --query
'TargetGroups[0].TargetGroupArn' --output text)

echo $TG_ARN
```

Register EC2 Instance to Target Group

```
aws elbv2 register-targets --target-group-arn $TG_ARN --targets
Id=$INSTANCE_ID
```

**Configure ALB**
Set up the ALB with a listener.

Get two subnets from different AZs

```
# Explicitly get one subnet from us-east-1a and one from us-east-1b

SUB_A=$(aws ec2 describe-subnets --region us-east-1 --filters
"Name=vpc-id,Values=$VPC_ID" "Name=availability-zone,Values=us-east-1a"
--query 'Subnets[0].SubnetId' --output text)

SUB_B=$(aws ec2 describe-subnets --region us-east-1 --filters
"Name=vpc-id,Values=$VPC_ID" "Name=availability-zone,Values=us-east-1b"
--query 'Subnets[0].SubnetId' --output text)

echo "Using Subnets: $SUB_A (1a) and $SUB_B (1b)"
```

Check the Instance's availability Zone

```
aws ec2 describe-instances --instance-ids $INSTANCE_ID --region us-east-1
--query
'Reservations[0].Instances[0].{Subnet:SubnetId,AZ:Placement.AvailabilityZon
e}'
```

If the instance's availability zone is not **1A** and **1B** then get the Subnet ID of the instance subnet then add it in the Load balancer.

**Create Application Load Balancer**

```
ALB_ARN=$(aws elbv2 create-load-balancer --name devops-alb --subnets $SUB_A
$SUB_B --security-groups $DEFAULT_SG_ID --query
'LoadBalancers[0].LoadBalancerArn' --output text)
```

Create Listener to route traffic

```
aws elbv2 create-listener --load-balancer-arn $ALB_ARN --protocol HTTP
--port 80 --default-actions Type=forward,TargetGroupArn=$TG_ARN
```

**Verification**

Get the ALB DNS Name

```
ALB_DNS=$(aws elbv2 describe-load-balancers --names devops-alb --query
'LoadBalancers[0].DNSName' --output text)

echo "Access your Nginx server at: http://$ALB_DNS"
```

# Day 37: Managing EC2 Access with S3 Role-based Permissions

## Task Description:

The Nautilus DevOps team needs to set up an application on an EC2 instance to interact with an S3 bucket for storing and retrieving data. To achieve this, the team must create a private S3 bucket, set appropriate IAM policies and roles, and test the application functionality.

1) **EC2 Instance Setup**:

- An instance named `xfusion-ec2` already exists.
- The instance requires access to an S3 bucket.

2) **Setup SSH Keys**:

- Create new SSH key pair (id_rsa and id_rsa.pub) on the `aws-client` host and add the public key to the `root` user's authorized keys on the EC2 instance.

3) **Create a Private S3 Bucket**:

- Name the bucket `xfusion-s3-23755`.
- Ensure the bucket is private.

4) **Create an IAM Policy and Role**:

- Create an IAM policy allowing `s3:PutObject`, `s3:ListBucket` and `s3:GetObject` access to `xfusion-s3-23755`.
- Create an IAM role named `xfusion-role`.
- Attach the policy to the IAM role.
- Attach this role to the `xfusion-ec2` instance.

5) **Test the Access**:

- SSH into the EC2 instance and try to upload a file to `xfusion-s3-23755` bucket using following command:

```
aws s3 cp <your-file> s3://xfusion-s3-23755/
```

- Now run following command to list the upload file:

```
aws s3 ls s3://xfusion-s3-23755/
```

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Set up Password-less SSH**

Generate Keys

```
ssh-keygen -t rsa -N "" -f /root/.ssh/id_rsa
```

Authorize Key

Get the Public IP of `xfusion-ec2` and add public key to the root user's `authorized_keys`.

```
EC2_IP=$(aws ec2 describe-instances --filters
"Name=tag:Name,Values=xfusion-ec2" --query
'Reservations[0].Instances[0].PublicIpAddress' --output text)
PUB_KEY=$(cat /root/.ssh/id_rsa.pub)

ssh -i <existing_key> ubuntu@$EC2_IP "sudo mkdir -p /root/.ssh && echo
'$PUB_KEY' | sudo tee -a /root/.ssh/authorized_keys"
```

**Create the Private S3 Bucket**
By default, buckets created via the CLI are private.

```
aws s3api create-bucket --bucket xfusion-s3-23755 --region us-east-1
```

**Create IAM Policy and Role**
The EC2 instance needs a "Role" to assume identity, and that role needs a "Policy" to grant permissions.

Create the Trust Policy (Allows EC2 to use this role):
Create Policy JSON file

```
touch trust-policy.json
chmod +x trust-policy.json
```

Add the content in the JSON file

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": { "Service": "ec2.amazonaws.com" },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Create Role

```
aws iam create-role --role-name xfusion-role --assume-role-policy-document
file://trust-policy.json
```

**Create the Access Policy**

```
touch s3-access-policy.json
chmod +x s3-access-policy.json
```

Add the content in the JSON file

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```

```
            "Action": ["s3:PutObject", "s3:GetObject", "s3:ListBucket"],
            "Resource": [
                "arn:aws:s3:::xfusion-s3-23755",
                "arn:aws:s3:::xfusion-s3-23755/*"
            ]
        }
    ]
}
```

Create policy

```
POLICY_ARN=$(aws iam create-policy --policy-name xfusion-s3-policy
--policy-document file://s3-access-policy.json --query 'Policy.Arn'
--output text)

echo $POLICY_ARN

aws iam attach-role-policy --role-name xfusion-role --policy-arn
$POLICY_ARN
```

**Attach Role to EC2 Instance**
To attach a role to an EC2 instance, we must use an **Instance Profile** as a container.

Create Instance Profile

```
aws iam create-instance-profile --instance-profile-name
xfusion-instance-profile
```

Add Role to Profile

```
aws iam add-role-to-instance-profile --instance-profile-name
xfusion-instance-profile --role-name xfusion-role
```

Associate with the EC2 Instance

```
INSTANCE_ID=$(aws ec2 describe-instances --filters
"Name=tag:Name,Values=xfusion-ec2" --query
'Reservations[0].Instances[0].InstanceId' --output text)

echo $INSTANCE_ID

aws ec2 associate-iam-instance-profile --instance-id $INSTANCE_ID
--iam-instance-profile Name=xfusion-instance-profile
```

**Test the Access**

Now, log into the instance and test the S3 operations.

```
ssh root@$EC2_IP

# Create a test file and upload it
echo "Hello from xfusion-ec2" > testfile.txt
aws s3 cp testfile.txt s3://xfusion-s3-23755/

# List the bucket to verify
aws s3 ls s3://xfusion-s3-23755/
```

# Day 38: Deploying Containerized Applications with Amazon ECS

## Task Description:

The Nautilus DevOps team is tasked with deploying a containerized application using Amazon's container services. They need to create a private Amazon Elastic Container Registry (ECR) to store their Docker images and use Amazon Elastic Container Service (ECS) to deploy the application. The process involves building a Docker image from a given Dockerfile, pushing it to the ECR, and then setting up an ECS cluster to run the application.

1. **Create a Private ECR Repository**:
    - Create a private ECR repository named `datacenter-ecr` to store Docker images.
2. **Build and Push Docker Image**:
    - Use the Dockerfile located at `/root/pyapp` on the `aws-client` host.
    - Build a Docker image using this Dockerfile.
    - Tag the image with `latest` tag.
    - Push the Docker image to the `datacenter-ecr` repository.
3. **Create and Configure ECS cluster**:
    - Create an ECS cluster named `datacenter-cluster` using the Fargate launch type.
4. **Create an ECS Task Definition**:
    - Define a task named `datacenter-taskdefinition` using the Docker image from the `datacenter-ecr` ECR repository.
    - Specify necessary CPU and memory resources.
5. **Deploy the Application Using ECS Service**:
    - Create a service named `datacenter-service` on the `datacenter-cluster` to run the task.
    - Ensure the service runs at least one task.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create the Private ECR Repository**

```
aws ecr create-repository --repository-name datacenter-ecr --region
us-east-1 --query 'repository.repositoryUri' --output text
```

**Note:** Save the output URI (e.g., `123456789012.dkr.ecr.us-east-1.amazonaws.com/datacenter-ecr`), as we will need it for tagging.

**ECR Authentication**

```
# get Account ID
ACCOUNT_ID=$(aws sts get-caller-identity --query 'Account' --output text)

echo $ACCOUNT_ID

# Authenticate with docker
aws ecr get-login-password --region us-east-1 | docker login --username AWS
--password-stdin ${ACCOUNT_ID}.dkr.ecr.us-east-1.amazonaws.com
```

**Build and Tag the Image**

Change directory to where the Dockerfile is located

```
cd /root/pyapp
```

Build the image locally

```
docker build -t datacenter-ecr:latest .
```

Set the ECR URI

```
ECR_URI="<ECR_URI>"
```

Tag the image for ECR repository

```
docker tag datacenter-ecr:latest $ECR_URI:latest
```

**Push the Image to ECR**

```
docker push $ECR_URI:latest
```

**Verify**

```
aws ecr list-images --repository-name datacenter-ecr --region us-east-1
```

When we use Fargate and pull images from a private ECR repository, Amazon ECS needs a **Task Execution Role**.

Even though Fargate is serverless, the "underlying agent" that pulls our image and sends logs to CloudWatch needs its own permissions. You must define an `executionRoleArn` in our JSON.

**Create the Execution Role**
If we haven't created the standard **ecsTaskExecutionRole** yet, run these commands:

Create the Trust Policy:

Create file

```
touch ecs-trust-policy.json
chmod +x ecs-trust-policy.json
```

Add the Following content in the file

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": { "Service": "ecs-tasks.amazonaws.com" },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Create the Role and Attach the Managed Policy:

```
aws iam create-role --role-name ecsTaskExecutionRole
--assume-role-policy-document file://ecs-trust-policy.json

aws iam attach-role-policy --role-name ecsTaskExecutionRole --policy-arn
arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy
```

**Configure the ECS Cluster**

Create a cluster that will manage Fargate tasks.

```
aws ecs create-cluster --cluster-name datacenter-cluster --region us-east-1
```

**Register the Task Definition**

Create a JSON file named `task-def.json` to define the Fargate task. We'll use standard Fargate values (0.25 vCPU and 0.5 GB RAM).

Create JSON file

```
touch task-def.json
chmod +x task-def.json
```

Add the following Content in the File

```
{
    "family": "datacenter-taskdefinition",
    "networkMode": "awsvpc",
    "executionRoleArn":
"arn:aws:iam::123456789012:role/ecsTaskExecutionRole",
    "containerDefinitions": [
        {
            "name": "pyapp-container",
            "image":
"123456789012.dkr.ecr.us-east-1.amazonaws.com/datacenter-ecr:latest",
            "portMappings": [
                {
                    "containerPort": 80,
                    "hostPort": 80,
                    "protocol": "tcp"
                }
            ],
            "essential": true
        }
    ],
```

```
    "requiresCompatibilities": [
        "FARGATE"
    ],
    "cpu": "256",
    "memory": "512"
}
```

Register it

```
aws ecs register-task-definition --cli-input-json file://task-def.json
--region us-east-1
```

**Deploy the ECS Service**

Finally, launch the service. We will need to provide a subnet and security group from VPC.

Get default VPC subnets

```
SUBNET_ID=$(aws ec2 describe-subnets --query 'Subnets[0].SubnetId' --output
text)
```

Get default VPC Security Group

```
SG_ID=$(aws ec2 describe-security-groups --filters
Name=group-name,Values=default --query 'SecurityGroups[0].GroupId' --output
text)
```

Allow Inbound HTTP (Port 80) from the internet

```
aws ec2 authorize-security-group-ingress --group-id $SG_ID --protocol tcp
--port 80 --cidr 0.0.0.0/0 --region us-east-1
```

**Create Service**

```
aws ecs create-service --cluster datacenter-cluster --service-name
datacenter-service --task-definition datacenter-taskdefinition
--desired-count 1 --launch-type FARGATE --network-configuration
"awsvpcConfiguration={subnets=[$SUBNET_ID],securityGroups=[$SG_ID],assignPu
blicIp=ENABLED}" --region us-east-1
```

**Verify**

```
aws ecs describe-services --cluster datacenter-cluster --services
datacenter-service --region us-east-1 --query "services[0].status"
```

# Day 39: Hosting a Static Website on AWS S3

## Task Description:

The Nautilus DevOps team has been tasked with creating an internal information portal for public access. As part of this project, they need to host a static website on AWS using an S3 bucket. The S3 bucket must be configured for public access to allow external users to access the static website directly via the S3 website URL.

Task Requirements:

1. Create an S3 bucket named `xfusion-web-6400`.
2. Configure the S3 bucket for static website hosting with `index.html` as the index document.
3. Allow public access to the bucket so that the website is publicly accessible.
4. Upload the `index.html` file from the `/root/` directory of the AWS client host to the S3 bucket.
5. Verify that the website is accessible directly through the S3 website URL.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create the S3 Bucket**

```
aws s3api create-bucket --bucket xfusion-web-6400 --region us-east-1
```

**Disable Block Public Access**

By default, S3 prevents public access at the bucket level.

```
aws s3api put-public-access-block --bucket xfusion-web-6400
--public-access-block-configuration
"BlockPublicAcls=false,IgnorePublicAcls=false,BlockPublicPolicy=false,Restr
ictPublicBuckets=false"
```

**Apply the Public Read Policy**

Create JSON File

```
touch website_policy.json
chmod +x website_policy.json
```

Add the following content in the file

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PublicReadGetObject",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::xfusion-web-6400/*"
        }
    ]
}
```

Attach the policy to the bucket

```
aws s3api put-bucket-policy --bucket xfusion-web-6400 --policy
file://website_policy.json
```

**Configure Static Website Hosting**

Enable the website feature

```
aws s3 website s3://xfusion-web-6400/ --index-document index.html
```

Upload the Website Content

```
aws s3 cp /root/index.html s3://xfusion-web-6400/index.html
```

**Verification**

```
echo "Website URL:
http://xfusion-web-6400.s3-website-us-east-1.amazonaws.com"

curl -I http://xfusion-web-6400.s3-website-us-east-1.amazonaws.com
```

# Day 40: Troubleshooting Internet Accessibility for an EC2-Hosted Application

## Task Description:

The Nautilus Development Team recently deployed a new web application hosted on an EC2 instance within a public VPC named `xfusion-vpc`. The application, running on an Nginx server, should be accessible from the internet on port 80. Despite configuring the security group `xfusion-sg` to allow traffic on port 80 and verifying the EC2 instance settings, the application remains inaccessible from the internet. The team suspects that the issue might be related to the VPC configuration, as all other components appear to be set up correctly. The DevOps team has been asked to troubleshoot and resolve the issue to ensure the application is accessible to external users.

As a member of the Nautilus DevOps Team, your task is to perform the following:

1. **Verify VPC Configuration:** Ensure that the VPC `xfusion-vpc` is properly configured to allow internet access.
2. **Ensure Accessibility:** Make sure the EC2 instance `xfusion-ec2` running the Nginx server is accessible from the internet on port 80.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)
***Notes:***
Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

**In this task, The Internet Gateway is not attached to the EC2 instance VPC, then we attached the Internet gateway to the VPC, the issue is resolved**

# Day 41: Securing Data with AWS KMS

## Task Description:

The Nautilus DevOps team is focusing on improving their data security by using AWS KMS. Your task is to create a KMS key and manage the encryption and decryption of a pre-existing sensitive file using the KMS key.

Specific Requirements:

1. Create a symmetric KMS key named `datacenter-KMS-Key` to manage encryption and decryption.
2. Encrypt the provided `SensitiveData.txt` file (located in /root/), base64 encode the ciphertext, and save the encrypted version as `EncryptedData.bin` in the `/root/` directory.
3. Try to decrypt the same and verify that the decrypted data matches the original file.

Make sure that the KMS key is correctly configured. The validation script will test your configuration by decrypting the `EncryptedData.bin` file using the KMS key you created.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

To secure sensitive data using AWS KMS, we need to create a **Customer Managed Key** (CMK), associate it with a friendly alias, and then use the AWS CLI to perform the cryptographic operations.

Using AWS CLI

**Create the KMS Key and Alias**
First, create a symmetric encryption key. Since the CLI `create-key` command doesn't accept a name directly, we must create the key first and then attach the alias `datacenter-KMS-Key`.

Create the symmetric KMS key and capture the KeyId

```
KEY_ID=$(aws kms create-key --description "Key for datacenter sensitive
data" --region us-east-1 --query 'KeyMetadata.KeyId' --output text)

echo $KEY_ID
```

Create the alias for the key

```
aws kms create-alias --alias-name alias/datacenter-KMS-Key --target-key-id
$KEY_ID --region us-east-1
```

**Encrypt the Sensitive File**
We will now encrypt `SensitiveData.txt`. The AWS CLI `encrypt` command returns a base64-encoded `CiphertextBlob` by default. To meet the requirement of saving the encrypted version as `EncryptedData.bin`, we must decode that blob into its binary form.

Encrypt the file and save the binary ciphertext to **EncryptedData.bin**

```
aws kms encrypt --key-id alias/datacenter-KMS-Key --plaintext
fileb:///root/SensitiveData.txt --output text --query CiphertextBlob |
base64 --decode > /root/EncryptedData.bin
```

**Decrypt and Verify**
To ensure the encryption worked correctly, decrypt the binary file and compare it to the original.

Decrypt the binary file back to plaintext

```
aws kms decrypt --ciphertext-blob fileb:///root/EncryptedData.bin --output
text --query Plaintext | base64 --decode > /root/DecryptedData.txt
```

Verify the contents match

```
diff /root/SensitiveData.txt /root/DecryptedData.txt && echo "Verification
Successful: Files match."
```

# Day 42: Building and Managing NoSQL Databases with AWS DynamoDB

## Task Description:

The Nautilus DevOps team is developing a simple 'To-Do' application using DynamoDB to store and manage tasks efficiently. The team needs to create a DynamoDB table to hold tasks, each identified by a unique task ID. Each task will have a description and a status, which indicates the progress of the task (e.g., 'completed' or 'in-progress').

Your task is to:

1. Create a DynamoDB table named `devops-tasks` with a primary key called `taskId` (string).
2. Insert the following tasks into the table:
    - Task 1: taskId: '1', description: 'Learn DynamoDB', status: 'completed'
    - Task 2: taskId: '2', description: 'Build To-Do App', status: 'in-progress'
3. Verify that Task 1 has a status of 'completed' and Task 2 has a status of 'in-progress'.

Ensure the DynamoDB table is created successfully and that both tasks are inserted correctly with the appropriate statuses.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

To implement the To-Do application's backend on DynamoDB, we need to create the schema with a single Partition Key and then populate it with the tasks.

DynamoDB is a NoSQL service, so while we define the `taskId` attribute during creation, the other attributes (`description` and `status`) are flexible and added during the item insertion phase.

Using AWS CLI

**Create the DynamoDB Table**
We will create a table named **devops-tasks** using the `taskId` as the String-type Partition Key. We'll use "On-Demand" billing to avoid managing throughput.

```
aws dynamodb create-table --table-name devops-tasks --attribute-definitions
AttributeName=taskId,AttributeType=S --key-schema
AttributeName=taskId,KeyType=HASH --billing-mode PAY_PER_REQUEST --region
us-east-1
```

**Note:** Wait a few seconds for the `TableStatus` to change from `CREATING` to `ACTIVE` before running the next commands.

Check Status

```
aws dynamodb describe-table --table-name devops-tasks --query
'Table.TableStatus'
```

**Insert Task Items**

**Task 1:**

```
aws dynamodb put-item --table-name devops-tasks \
    --item '{
        "taskId": {"S": "1"},
        "description": {"S": "Learn DynamoDB"},
        "status": {"S": "completed"}
    }' \
    --region us-east-1
```

**Task 2:**

```
aws dynamodb put-item --table-name devops-tasks \
    --item '{
        "taskId": {"S": "2"},
        "description": {"S": "Build To-Do App"},
        "status": {"S": "in-progress"}
```

```
    }' \
    --region us-east-1
```

**Verify**

To verify the insertion and statuses, use the `get-item` command for each task.

```
# Verify Task 1
aws dynamodb get-item --table-name devops-tasks --key '{"taskId": {"S": "1"}}' --region us-east-1

# Verify Task 2
aws dynamodb get-item --table-name devops-tasks --key '{"taskId": {"S": "2"}}' --region us-east-1
```

# Day 43: Scaling and Managing Kubernetes Clusters with Amazon EKS

## Task Description:

The Nautilus DevOps team has been tasked with preparing the infrastructure for a new Kubernetes-based application that will be deployed using Amazon EKS. The team is in the process of setting up an EKS cluster that meets their internal security and scalability standards. They require that the cluster be provisioned using the latest stable Kubernetes version to take advantage of new features and security improvements.

To minimize external exposure, the EKS cluster endpoint must be kept private. Additionally, the cluster needs to use the default VPC with availability zones `a`, `b`, and `c` to ensure high availability across different physical locations.

Your task is to create an EKS cluster named `devops-eks`, along with an IAM role for the cluster named `eksClusterRole`. The Kubernetes version must be `1.30`. Ensure that the cluster endpoint access is configured as private.

Finally, verify that the EKS cluster is successfully created with the correct configuration and is ready for workloads.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)
***Notes:***
Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create the EKS Cluster IAM Role**

Create the Trust Policy

```
touch eks-trust-policy.json
chmod +x eks-trust-policy.json
```

Add the following content in the file

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": { "Service": "eks.amazonaws.com" },
      "Action": [
        "sts:AssumeRole",
        "sts:TagSession"
      ]
    }
  ]
}
```

Create the Role and Attach Policy

```
aws iam create-role --role-name eksClusterRole
--assume-role-policy-document file://eks-trust-policy.json

# Attached Policies

aws iam attach-role-policy --role-name eksClusterRole --policy-arn
arn:aws:iam::aws:policy/AmazonEKSClusterPolicy

aws iam attach-role-policy --role-name eksClusterRole --policy-arn
arn:aws:iam::aws:policy/AmazonEKSComputePolicy

aws iam attach-role-policy --role-name eksClusterRole --policy-arn
arn:aws:iam::aws:policy/AmazonEKSBlockStoragePolicy

aws iam attach-role-policy --role-name eksClusterRole --policy-arn
```

```
arn:aws:iam::aws:policy/AmazonEKSLoadBalancingPolicy

aws iam attach-role-policy --role-name eksClusterRole --policy-arn
arn:aws:iam::aws:policy/AmazonEKSNetworkingPolicy
```

**Identify Default VPC Subnets**
To ensure high availability across AZs a, b, and c, we must retrieve the subnet IDs from default VPC associated with those zones.

Get the Default VPC ID

```
VPC_ID=$(aws ec2 describe-vpcs --filters "Name=isDefault,Values=true"
--query 'Vpcs[0].VpcId' --output text)

echo $VPC_ID
```

Get Subnets for **us-east-1a**, **us-east-1b**, and **us-east-1c**

```
aws ec2 describe-subnets --filters "Name=vpc-id,Values=$VPC_ID" --query
'Subnets[?AvailabilityZone==`us-east-1a` || AvailabilityZone==`us-east-1b`
|| AvailabilityZone==`us-east-1c`].SubnetId' --output text
```

Note the Subnet IDs.

**Create the EKS Cluster**
create the cluster using Kubernetes version **1.30** and disable public endpoint access.

Get Role ARN

```
ROLE_ARN=$(aws iam get-role --role-name eksClusterRole --query 'Role.Arn'
--output text)

echo $ROLE_ARN
```

Create Cluster

```
aws eks create-cluster --name devops-eks --kubernetes-version 1.30
--role-arn $ROLE_ARN --resources-vpc-config
subnetIds=<Subnet1>,<Subnet2>,<Subnet3>,endpointPrivateAccess=true,endpoint
PublicAccess=false
```

**Verify**

EKS cluster creation typically takes 10–15 minutes.

```
aws eks describe-cluster --name devops-eks --query "cluster.{Status:status,
Version:version, EndpointAccess:resourcesVpcConfig.endpointPrivateAccess}"
```

# Day 44: Implementing Auto Scaling for High Availability in AWS

## Task Description:

The DevOps team is tasked with setting up a highly available web application using AWS. To achieve this, they plan to use an Auto Scaling Group (ASG) to ensure that the required number of EC2 instances are always running, and an Application Load Balancer (ALB) to distribute traffic across these instances. The goal of this task is to set up an ASG that automatically scales EC2 instances based on **CPU utilization**, and an ALB that directs incoming traffic to the instances. The EC2 instances should have Nginx installed and running to serve web traffic.

1. Create an EC2 launch template named `xfusion-launch-template` that specifies the configuration for the EC2 instances, including the **Amazon Linux 2 AMI**, **t2.micro** instance type, and a security group that allows **HTTP traffic on port 80**.
2. Add a **User Data** script to the launch template to install Nginx on the EC2 instances when they are launched. The script should install Nginx, start the Nginx service, and enable it to start on boot.
3. Create an Auto Scaling Group named `xfusion-asg` that uses the launch template and ensures a minimum of **1 instance**, desired capacity is **1 instance** and a maximum of **2 instances** are running based on **CPU utilization**. Set the target **CPU utilization** to **50%**.
4. Create a target group named `xfusion-tg`, an Application Load Balancer named `xfusion-alb` and configure it to listen on **port 80**. Ensure the ALB is associated with the Auto Scaling Group and distributes traffic across the instances.
5. Configure health checks on the ALB to ensure it routes traffic only to healthy instances.
6. Verify that the ALB's DNS name is accessible and that it displays the default Nginx page served by the EC2 instances.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create a Security Group**
Before creating the Launch Template, we need a security group that allows port 80 traffic.

Get the Default VPC ID

```
VPC_ID=$(aws ec2 describe-vpcs --region us-east-1 --filters
"Name=isDefault,Values=true" --query 'Vpcs[0].VpcId' --output text)
```

Create the Security Group

```
SG_ID=$(aws ec2 create-security-group --group-name xfusion-sg --region
us-east-1 --description "Allow HTTP" --vpc-id $VPC_ID --query 'GroupId'
--output text)
```

Add Inbound Rule for Port 80

```
aws ec2 authorize-security-group-ingress --group-id <Security-Group_ID>
--protocol tcp --port 80 --cidr 0.0.0.0/0
```

**Create the Launch Template**
The Launch Template contains the "blueprint" for our instances, including the User Data script to install Nginx.

**Prepare the Configuration Script (User Data)**
Create the script file

```
touch userdata.sh
chmod +x userdata.sh
```

Add the following content in userdata.sh file

```
#!/bin/bash

sudo yum update -y
```

```
sudo amazon-linux-extras install nginx1 -y
sudo systemctl start nginx
sudo systemctl enable nginx
```

**Retrieve the Latest AMI ID using SSM Parameter Store**

```
AMI_ID=$(aws ssm get-parameter --region us-east-1 --name
/aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-x86_64-gp2 --query
'Parameter.Value' --output text)

echo "Using AMI ID: $AMI_ID"
```

**Create the Template**

```
aws ec2 create-launch-template --launch-template-name
xfusion-launch-template --launch-template-data "{
        \"ImageId\":\"$AMI_ID\",
        \"InstanceType\":\"t2.micro\",
        \"SecurityGroupIds\":[\"$SG_ID\"],
        \"UserData\":\"$(base64 -w 0 userdata.sh)\"
    }" --region us-east-1
```

**Set up the ALB and Target Group**

Create the Target Group

```
TG_ARN=$(aws elbv2 create-target-group --name xfusion-tg --protocol HTTP
--port 80 --vpc-id $VPC_ID --target-type instance --region us-east-1
--query 'TargetGroups[0].TargetGroupArn' --output text)

echo $TG_ARN
```

Retrieve at least two subnets from default VPC first

```
# Explicitly get one subnet from us-east-1a and one from us-east-1b

SUB_A=$(aws ec2 describe-subnets --region us-east-1 --filters
"Name=vpc-id,Values=$VPC_ID" "Name=availability-zone,Values=us-east-1a"
--query 'Subnets[0].SubnetId' --output text)

SUB_B=$(aws ec2 describe-subnets --region us-east-1 --filters
"Name=vpc-id,Values=$VPC_ID" "Name=availability-zone,Values=us-east-1b"
--query 'Subnets[0].SubnetId' --output text)

echo "Using Subnets: $SUB_A (1a) and $SUB_B (1b)"
```

Create ALB

```
ALB_DATA=$(aws elbv2 create-load-balancer --name xfusion-alb --region
us-east-1 --subnets $SUB_A $SUB_B --security-groups $SG_ID --query
'LoadBalancers[0].[LoadBalancerArn,DNSName]' --output text)
```

Get Load Balancer DNS and ARN

```
ALB_ARN=$(echo $ALB_DATA | awk '{print $1}')
ALB_DNS=$(echo $ALB_DATA | awk '{print $2}')
```

Create Listener

```
aws elbv2 create-listener --load-balancer-arn $ALB_ARN --protocol HTTP
--port 80 --default-actions Type=forward,TargetGroupArn=$TG_ARN
```

**Create the Auto Scaling Group (ASG)**
Now, create the ASG and link it to the Target Group so it automatically registers new instances.

Create ASG:

```
aws autoscaling create-auto-scaling-group --auto-scaling-group-name
xfusion-asg --launch-template LaunchTemplateName=xfusion-launch-template
--target-group-arns $TG_ARN --min-size 1 --max-size 2 --desired-capacity 1
--vpc-zone-identifier "$(echo $SUBNETS | tr ' ' ',')"
```

Add Scaling Policy (50% CPU Target)

```
aws autoscaling put-scaling-policy --auto-scaling-group-name xfusion-asg
--policy-name cpu-scaling-policy --policy-type TargetTrackingScaling
--target-tracking-configuration "{
        \"PredefinedMetricSpecification\": { \"PredefinedMetricType\":
\"ASGAverageCPUUtilization\" },
        \"TargetValue\": 50.0
    }"
```

**Verification**

```
echo "Access website at: http://$ALB_DNS"
curl -I http://$ALB_DNS
```

# Day 45: Configure NAT Gateway for Internet Access in a Private VPC

## Task Description:

The Nautilus DevOps team is tasked with enabling internet access for an EC2 instance running in a private subnet. This instance should be able to upload a test file to a public S3 bucket once it can access the internet. To achieve this, the team must set up a NAT Gateway in a public subnet within the same VPC.

1) A VPC named `datacenter-priv-vpc` and a private subnet `datacenter-priv-subnet` have already been created.
2) An EC2 instance named `datacenter-priv-ec2` is already running in the private subnet.
3) The EC2 instance is configured with a cron job that uploads a test file to a bucket `datacenter-nat-16391` once the internet is accessible.

Your task is to:

- Create a public subnet named `datacenter-pub-subnet` in the same VPC.
- Create an Internet Gateway and attach it to the VPC.
- Create a route table `datacenter-pub-rt` and associate it with the public subnet.
- Allocate an Elastic IP and create a NAT Gateway named `datacenter-natgw`.
- Update the private route table to route 0.0.0.0/0 traffic via the NAT Gateway.

Once complete, verify that the EC2 instance can reach the internet by confirming the presence of the test file in the S3 bucket `datacenter-nat-16391`. After completing all the configuration, please wait a few minutes for the test file to appear in the bucket, as it may take `2–3 minutes`.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)
***Notes:***
Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create the Public Subnet**

Get the VPC ID of **datacenter-priv-vpc**

```
VPC_ID=$(aws ec2 describe-vpcs --filters
"Name=tag:Name,Values=datacenter-priv-vpc" --query 'Vpcs[0].VpcId' --output
text)
```

Create the public subnet (adjust CIDR to avoid overlap, e.g., 10.0.1.0/24)

```
PUB_SUBNET_ID=$(aws ec2 create-subnet --vpc-id $VPC_ID --cidr-block <CIDR>
--availability-zone us-east-1a --tag-specifications
'ResourceType=subnet,Tags=[{Key=Name,Value=datacenter-pub-subnet}]' --query
'Subnet.SubnetId' --output text)

echo $PUB_SUBNET_ID
```

**Create and Attach the Internet Gateway (IGW)**

Create the IGW

```
IGW_ID=$(aws ec2 create-internet-gateway --query
'InternetGateway.InternetGatewayId' --output text)
```

Attach it to the VPC

```
aws ec2 attach-internet-gateway --internet-gateway-id $IGW_ID --vpc-id
$VPC_ID
```

**Configure the Public Route Table**

Create route table

```
PUB_RT_ID=$(aws ec2 create-route-table --vpc-id $VPC_ID
--tag-specifications
'ResourceType=route-table,Tags=[{Key=Name,Value=datacenter-pub-rt}]'
--query 'RouteTable.RouteTableId' --output text)
```

Add route to Internet Gateway

```
aws ec2 create-route --route-table-id $PUB_RT_ID --destination-cidr-block
0.0.0.0/0 --gateway-id $IGW_ID
```

Associate with public subnet

```
aws ec2 associate-route-table --subnet-id $PUB_SUBNET_ID --route-table-id
$PUB_RT_ID
```

Get the Private Subnet ID

```
PRIV_SUBNET_ID=$(aws ec2 describe-subnets --filters
"Name=tag:Name,Values=datacenter-priv-subnet" --query "Subnets[0].SubnetId"
--output text)

echo $PRIV_SUBNET_ID
```

Get Private Route Table ID

```
PRIV_RT_ID=$(aws ec2 describe-route-tables --filters
"Name=tag:Name,Values=datacenter-priv-rt" --query
"RouteTables[0].RouteTableId" --output text)

echo $PRIV_RT_ID
```

Associate with private subnet with the private route table

```
aws ec2 associate-route-table --subnet-id $PRIV_SUBNET_ID --route-table-id
$PRIV_RT_ID
```

**Create the NAT Gateway**

A NAT Gateway requires a static **Elastic IP (EIP)** and must be placed in the **public** subnet.

Allocate Elastic IP

```
EIP_ALLOC=$(aws ec2 allocate-address --domain vpc --query 'AllocationId'
--output text)
```

Create NAT Gateway in the Public Subnet

```
NAT_GW_ID=$(aws ec2 create-nat-gateway --subnet-id $PUB_SUBNET_ID
--allocation-id $EIP_ALLOC --tag-specifications
'ResourceType=natgateway,Tags=[{Key=Name,Value=datacenter-natgw}]' --query
'NatGateway.NatGatewayId' --output text)
```

**Update the Private Route Table**
Tell the **private** subnet to send its outbound traffic to the NAT Gateway.

Add route to the NAT Gateway

```
aws ec2 create-route --route-table-id $PRIV_RT_ID --destination-cidr-block
0.0.0.0/0 --nat-gateway-id $NAT_GW_ID
```

**Verification**
Verify the file in S3

```
aws s3 ls s3://datacenter-nat-16391/
```

# Day 46: Event-Driven Processing with Amazon S3 and Lambda

## Task Description:

The DevOps team is working on automating file management between two S3 buckets. The task is to create a public S3 bucket for file uploads and a private S3 bucket for securely storing the files. A Lambda function will be triggered automatically whenever a file is uploaded to the public S3 bucket, which will copy the file to the private bucket. Additionally, logs of the operation will be stored in a DynamoDB table. The logs should include details such as the source bucket, destination bucket, and the object key of the file that was copied. This will help the team maintain better security and visibility for file transfers.

1. Create a public S3 bucket named `devops-public-2524`. Ensure that the bucket allows public access to its objects.
2. Create a private S3 bucket named `devops-private-26228`. Ensure that the bucket does not allow public access.
3. Create a Lambda function named `devops-copyfunction`. This function should be triggered by uploads to the public S3 bucket and should copy the uploaded file to the private bucket. Create the necessary policies and a role named `lambda_execution_role`. Attach these policies to the role, and then link this role to the Lambda function.
4. `lambda-function.py` is already present under the `/root/` directory on AWS client host, replace `REPLACE-WITH-YOUR-DYNAMODB-TABLE` and `REPLACE-WITH-YOUR-PRIVATE-BUCKET` values.
5. Create a DynamoDB table named `devops-S3CopyLogs` with a partition key `LogID` (string). This table will store logs generated by the Lambda function, including details such as source bucket name, destination bucket name, and object key.
6. For testing upload the file `sample.zip` located in the `/root` directory on the client host to the public S3 bucket. The Lambda function should trigger and copy the file to the private bucket.
7. Verify that the file has been successfully copied to the private bucket by checking the private bucket in the S3 console.
8. Verify that a log entry has been created in the DynamoDB table containing the file copy details.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create the S3 Buckets**

Public Bucket

```
aws s3api create-bucket --bucket devops-public-2524 --region us-east-1

# Disable "Block Public Access" to allow public object ACLs
aws s3api delete-public-access-block --bucket devops-public-2524
```

Private Bucket

```
aws s3api create-bucket --bucket devops-private-26228 --region us-east-1
```

**Create the DynamoDB Table**
DynamoDB table stores transfer logs

```
aws dynamodb create-table --table-name devops-S3CopyLogs
--attribute-definitions AttributeName=LogID,AttributeType=S --key-schema
AttributeName=LogID,KeyType=HASH --provisioned-throughput
ReadCapacityUnits=5,WriteCapacityUnits=5 --region us-east-1
```

**Setup IAM Role and Policies**
The Lambda function needs a role to read from one bucket, write to another, and log to DynamoDB.

Create Trust Policy File

```
touch trust-policy.json
chmod +x trust-policy.json
```

Add the following content in the file

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Principal": { "Service": "lambda.amazonaws.com" },
    "Action": "sts:AssumeRole"
  }]
}
```

Create the Role

```
aws iam create-role --role-name lambda_execution_role
--assume-role-policy-document file://trust-policy.json
```

Attach Required Permissions

```
# Basic Execution (CloudWatch Logs)
aws iam attach-role-policy --role-name lambda_execution_role --policy-arn
arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole

# S3 Access
aws iam attach-role-policy --role-name lambda_execution_role --policy-arn
arn:aws:iam::aws:policy/AmazonS3FullAccess

# DynamoDB Access
aws iam attach-role-policy --role-name lambda_execution_role --policy-arn
arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess
```

**Deploy the Lambda Function**

**Prepare the Code:** Edit `lambda-function.py` and replace the placeholders with `devops-S3CopyLogs` and `devops-private-26228`.

**lambda-function.py file**

```
import json
import boto3
from datetime import datetime
import uuid
```

```
# Initialize the S3 and DynamoDB clients
s3 = boto3.client('s3')
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('datacenter-S3CopyLogs')

def lambda_handler(event, context):
    try:
        # Get the source bucket and object key from the event
        source_bucket = event['Records'][0]['s3']['bucket']['name']
        object_key = event['Records'][0]['s3']['object']['key']

        # Hardcoded destination bucket name
        destination_bucket = "datacenter-private-15364"

        # Log the event details for debugging
        print(f"[INFO] Source bucket: {source_bucket}, Object key:
{object_key}")
        print(f"[INFO] Destination bucket: {destination_bucket}")

        # Copy the file from source bucket to destination bucket
        copy_source = {
            'Bucket': source_bucket,
            'Key': object_key
        }

        print(f"[INFO] Attempting to copy object from
{source_bucket}/{object_key} to {destination_bucket}/{object_key}")
        s3.copy_object(
            CopySource=copy_source,
            Bucket=destination_bucket,
            Key=object_key
        )
        print(f"[INFO] File successfully copied from
{source_bucket}/{object_key} to {destination_bucket}/{object_key}")

        # Create log entry for DynamoDB
        log_entry = {
            'LogID': str(uuid.uuid4()),  # Generate a unique ID for the log
entry
            'SourceBucket': source_bucket,
            'DestinationBucket': destination_bucket,
            'ObjectKey': object_key,
```

```
            'Timestamp': datetime.now().strftime('%Y-%m-%d %H:%M:%S'),
            'Status': 'Success'
        }

        # Log the log entry before attempting to write to DynamoDB
        print(f"[INFO] Writing the following log entry to
DynamoDB:\n{json.dumps(log_entry, indent=4)}")
        table.put_item(Item=log_entry)
        print(f"[INFO] Successfully wrote log entry to DynamoDB")

        return {
            'statusCode': 200,
            'body': json.dumps(f"File successfully copied to
{destination_bucket}")
        }

    except Exception as e:
        # Store error log in DynamoDB in case of failure
        log_entry = {
            'LogID': str(uuid.uuid4()),  # Generate a unique ID for the log
entry
            'SourceBucket': source_bucket,
            'DestinationBucket': destination_bucket,
            'ObjectKey': object_key,
            'Timestamp': datetime.now().strftime('%Y-%m-%d %H:%M:%S'),
            'Status': 'Failure',
            'Error': str(e)
        }

        # Log the error log entry before attempting to write to DynamoDB
        print(f"[ERROR] Writing the following error log entry to
DynamoDB:\n{json.dumps(log_entry, indent=4)}")
        try:
            table.put_item(Item=log_entry)
            print(f"[INFO] Successfully wrote error log entry to DynamoDB")
        except Exception as db_error:
            print(f"[ERROR] Failed to write error log entry to DynamoDB:
{str(db_error)}")

        # Log the error in CloudWatch
        print(f"[ERROR] Error during file copy or DynamoDB operation:
{str(e)}")
        return {
```

```
            'statusCode': 500,
            'body': json.dumps(f"Error copying file: {str(e)}")
        }
```

Package and Create

```
zip function.zip lambda-function.py

LAMBDA_ARN=$(aws lambda create-function --function-name devops-copyfunction
--runtime python3.12 --role arn:aws:iam::$(aws sts get-caller-identity
--query Account --output text):role/lambda_execution_role --handler
lambda-function.lambda_handler --zip-file fileb://function.zip --region
us-east-1 --query 'FunctionArn' --output text)
```

**Configure the S3 Trigger**

We must give S3 permission to invoke our function and then set the notification.

Add Permission

```
aws lambda add-permission --function-name devops-copyfunction
--statement-id s3-trigger --action lambda:InvokeFunction --principal
s3.amazonaws.com --source-arn arn:aws:s3:::devops-public-2524
```

Add Trigger Configuration

Create the notification.json file

```
touch notification.json
chmod +x notification.json
```

Add the following content in the file

```
{
  "LambdaFunctionConfigurations": [{
    "LambdaFunctionArn": "REPLACE_WITH_LAMBDA_ARN",
    "Events": ["s3:ObjectCreated:*"]
  }]
}
```

Replace `REPLACE_WITH_LAMBDA_ARN` with the actual ARN from the previous step and run:

```
aws s3api put-bucket-notification-configuration --bucket devops-public-2524
```

```
--notification-configuration file://notification.json
```

**Verification**

Upload the test file

```
aws s3 cp /root/sample.zip s3://devops-public-2524/
```

# Day 47: Integrating AWS SQS and SNS for Reliable Messaging

## Task Description:

The Nautilus DevOps team needs to implement priority queuing using Amazon SQS and SNS. The goal is to create a system where messages with different priorities are handled accordingly. You are required to use AWS CloudFormation to deploy the necessary resources in your AWS account. The CloudFormation template should be created on the AWS client host at `/root/xfusion-priority-stack.yml`, the stack name must be `xfusion-priority-stack` and it should create the following resources:

1. Two SQS queues named `xfusion-High-Priority-Queue` and `xfusion-Low-Priority-Queue`.
2. An SNS topic named `xfusion-Priority-Queues-Topic`.
3. A Lambda function named `xfusion-priorities-queue-function` that will consume messages from the SQS queues. The Lambda function code is provided in `/root/index.py` on the AWS client host.
4. An IAM role named `lambda_execution_role` that provides the necessary permissions for the Lambda function to interact with SQS and SNS.

Once the stack is deployed, to test the same you can publish messages to the SNS topic, invoke the Lambda function and observe the order in which they are processed by the Lambda function. The high-priority message must be processed first.

```
topicarn=$(aws sns list-topics --query "Topics[?contains(TopicArn,
'xfusion-Priority-Queues-Topic')].TopicArn" --output text)

aws sns publish --topic-arn $topicarn --message 'High Priority message 1'
--message-attributes '{"priority" : { "DataType":"String",
"StringValue":"high"}}'

aws sns publish --topic-arn $topicarn --message 'High Priority message 2'
--message-attributes '{"priority" : { "DataType":"String",
"StringValue":"high"}}'

aws sns publish --topic-arn $topicarn --message 'Low Priority message 1'
--message-attributes '{"priority" : { "DataType":"String",
"StringValue":"low"}}'

aws sns publish --topic-arn $topicarn --message 'Low Priority message 2'
```

```
--message-attributes '{"priority" : { "DataType":"String",
"StringValue":"low"}}'
```

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create the CloudFormation Template**

create file `xfusion-priority-stack.yml`

```
touch xfusion-priority-stack.yml
chmod +x xfusion-priority-stack.yml
```

Add the following content in the file

```
AWSTemplateFormatVersion: '2010-09-09'
Description: SQS Priority Queuing Stack with SNS and Lambda

Resources:
  HighPriorityQueue:
    Type: AWS::SQS::Queue
    Properties:
```

```
    QueueName: xfusion-High-Priority-Queue
    VisibilityTimeout: 60

LowPriorityQueue:
  Type: AWS::SQS::Queue
  Properties:
    QueueName: xfusion-Low-Priority-Queue
    VisibilityTimeout: 60

PriorityQueuesTopic:
  Type: AWS::SNS::Topic
  Properties:
    TopicName: xfusion-Priority-Queues-Topic

# SNS Subscriptions with Filter Policies
HighPrioritySubscription:
  Type: AWS::SNS::Subscription
  Properties:
    TopicArn: !Ref PriorityQueuesTopic
    Protocol: sqs
    Endpoint: !GetAtt HighPriorityQueue.Arn
    FilterPolicy:
      priority:
        - high

LowPrioritySubscription:
  Type: AWS::SNS::Subscription
  Properties:
    TopicArn: !Ref PriorityQueuesTopic
    Protocol: sqs
    Endpoint: !GetAtt LowPriorityQueue.Arn
    FilterPolicy:
      priority:
        - low

# SQS Queue Policies to allow SNS to send messages
HighPriorityQueuePolicy:
  Type: AWS::SQS::QueuePolicy
  Properties:
    Queues:
      - !Ref HighPriorityQueue
    PolicyDocument:
      Statement:
```

```
        - Effect: Allow
          Principal: "*"
          Action: SQS:SendMessage
          Resource: !GetAtt HighPriorityQueue.Arn
          Condition:
            ArnEquals:
              aws:SourceArn: !Ref PriorityQueuesTopic

LowPriorityQueuePolicy:
  Type: AWS::SQS::QueuePolicy
  Properties:
    Queues:
      - !Ref LowPriorityQueue
    PolicyDocument:
      Statement:
        - Effect: Allow
          Principal: "*"
          Action: SQS:SendMessage
          Resource: !GetAtt LowPriorityQueue.Arn
          Condition:
            ArnEquals:
              aws:SourceArn: !Ref PriorityQueuesTopic

LambdaExecutionRole:
  Type: AWS::IAM::Role
  Properties:
    RoleName: lambda_execution_role
    AssumeRolePolicyDocument:
      Version: '2012-10-17'
      Statement:
        - Effect: Allow
          Principal:
            Service: lambda.amazonaws.com
          Action: sts:AssumeRole
    ManagedPolicyArns:
      - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
      - arn:aws:iam::aws:policy/AmazonSQSFullAccess
      - arn:aws:iam::aws:policy/AmazonSNSFullAccess

PriorityLambdaFunction:
  Type: AWS::Lambda::Function
  Properties:
    FunctionName: xfusion-priorities-queue-function
```

```
      Handler: index.lambda_handler
      Runtime: python3.9
      Role: !GetAtt LambdaExecutionRole.Arn
      Environment:
        Variables:
          high_priority_queue: !Ref HighPriorityQueue
          low_priority_queue: !Ref LowPriorityQueue
      Code:
        ZipFile: |
          import boto3
          import os

          sqs = boto3.client('sqs')
          def delete_message(queue_url, receipt_handle, message):
              response = sqs.delete_message(QueueUrl=queue_url,
ReceiptHandle=receipt_handle)
              return "Message " + "'" + message + "'" + " deleted"

          def poll_messages(queue_url):
              QueueUrl=queue_url
              response = sqs.receive_message(
                  QueueUrl=QueueUrl,
                  AttributeNames=[],
                  MaxNumberOfMessages=1,
                  MessageAttributeNames=['All'],
                  WaitTimeSeconds=3
              )

              if "Messages" in response:
                  receipt_handle=response['Messages'][0]['ReceiptHandle']
                  message = response['Messages'][0]['Body']
                  delete_response =
delete_message(QueueUrl,receipt_handle,message)
                  return delete_response
              else:
                  return "No more messages to poll"

          def lambda_handler(event, context):
              response = poll_messages(os.environ['high_priority_queue'])
              if response == "No more messages to poll":
                  response =
poll_messages(os.environ['low_priority_queue'])
              return response
```

**Deploy the Stack**

Run the following command to deploy the infrastructure

```
aws cloudformation create-stack --stack-name xfusion-priority-stack
--template-body file:///root/xfusion-priority-stack.yml --capabilities
CAPABILITY_NAMED_IAM --region us-east-1
```

**Test Priority Processing**

Once the stack status is `CREATE_COMPLETE`, run test commands to publish messages with attributes.

Get Topic ARN

```
topicarn=$(aws sns list-topics --query "Topics[?contains(TopicArn,
'xfusion-Priority-Queues-Topic')].TopicArn" --output text)
```

Publish High and Low priority messages

```
aws sns publish --topic-arn $topicarn --message 'High Priority message 1'
--message-attributes '{"priority" : { "DataType":"String",
"StringValue":"high"}}'

aws sns publish --topic-arn $topicarn --message 'Low Priority message 1'
--message-attributes '{"priority" : { "DataType":"String",
"StringValue":"low"}}'
```

**Verification**

Get URL for the High Priority Queue

```
HIGH_URL=$(aws sqs get-queue-url --queue-name xfusion-High-Priority-Queue
--query 'QueueUrl' --output text)
```

Get URL for the Low Priority Queue

```
LOW_URL=$(aws sqs get-queue-url --queue-name xfusion-Low-Priority-Queue
--query 'QueueUrl' --output text)
```

**SQS Check:** Verify messages are in the correct queues using

Check High Priority Queue

```
aws sqs get-queue-attributes --queue-url $HIGH_URL --attribute-names
ApproximateNumberOfMessages ApproximateNumberOfMessagesNotVisible
```

Check Low Priority Queue

```
aws sqs get-queue-attributes --queue-url $LOW_URL --attribute-names All
```

**Lambda Logs:** Invoke the Lambda function and check the CloudWatch logs. We should see the code polling the `HIGH_QUEUE_URL` first before checking the `LOW_QUEUE_URL`.

# Day 48: Automating Infrastructure Deployment with AWS CloudFormation

## Task Description:

The Nautilus DevOps team needs to implement a Lambda function using a CloudFormation stack. Create a CloudFormation template named `/root/xfusion-lambda.yml` on the AWS client host and configure it to create the following components. The stack name must be `xfusion-lambda-app`.

1. Create a Lambda function named `xfusion-lambda`.
2. Use the Runtime `Python`.
3. The function should print the body `Welcome to KKE AWS Labs!`.
4. Ensure the status code is `200`.
5. Create and use the IAM role named `lambda_execution_role`.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)
***Notes:***
Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create CloudFormation Template**
Create the xfusion-lambda.yml file

```
touch xfusion-lambda.yml
chmod +x xfusion-lambda.yml
```

Add the following content in the file

```
AWSTemplateFormatVersion: '2010-09-09'
Description: CloudFormation stack for xfusion-lambda-app

Resources:
  # IAM Role for Lambda execution
  LambdaExecutionRole:
    Type: AWS::IAM::Role
    Properties:
      RoleName: lambda_execution_role
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - lambda.amazonaws.com
            Action:
              - sts:AssumeRole
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole

  # Lambda Function
  XFusionLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: xfusion-lambda
      Handler: index.lambda_handler
      Runtime: python3.12
      Role: !GetAtt LambdaExecutionRole.Arn
      Code:
        ZipFile: |
          import json
          def lambda_handler(event, context):
              print("Welcome to KKE AWS Labs!")
              return {
                  'statusCode': 200,
                  'body': json.dumps('Welcome to KKE AWS Labs!')
              }
```

**Deploy the stack**

Run the following command to create the stack in the `us-east-1` region. Note the use of `--capabilities CAPABILITY_NAMED_IAM`, which is required because we are creating a role with a specific name.

```
aws cloudformation create-stack --stack-name xfusion-lambda-app
--template-body file:///root/xfusion-lambda.yml --capabilities
CAPABILITY_NAMED_IAM --region us-east-1
```

**Verification**

Once the stack status is `CREATE_COMPLETE`, We can test the function by invoking it via the CLI:

```
aws lambda invoke --function-name xfusion-lambda --region us-east-1
output.json

cat output.json
```

# Day 49: Centralized Audit Logging with VPC Peering

## Task Description:

The Nautilus DevOps team needs to build a secure and scalable log aggregation setup within their AWS environment. The goal is to gather log files from an internal EC2 instance running in a private VPC, transfer them securely to another EC2 instance in a public VPC, and then push those logs to a secure S3 bucket.

A VPC named `datacenter-priv-vpc` already exists with a private subnet named `datacenter-priv-subnet`, a route table named `datacenter-priv-rt`, and an EC2 instance named `datacenter-priv-ec2` (using `ubuntu` image). This instance uses the SSH key pair `datacenter-key.pem` already available on the AWS client host at `/root/.ssh/`.

Your task is to:

- Create a new VPC named `datacenter-pub-vpc`.
- Create a subnet named `datacenter-pub-subnet` and a route table named `datacenter-pub-rt` under this public VPC.
- Attach an internet gateway to `datacenter-pub-vpc` and configure the public route table to enable internet access.
- Launch an EC2 instance named `datacenter-pub-ec2` into the public subnet using the same key pair as the private instance.
- Create an IAM role named `datacenter-s3-role` with `PutObject` permission to an S3 bucket and attach it to the public EC2 instance.
- Create a new private S3 bucket named `datacenter-s3-logs-17209`.
- Configure a VPC Peering named `datacenter-vpc-peering` between the private and public VPCs.
- Modify both `datacenter-priv-rt` and `datacenter-pub-rt` to route each other's CIDR blocks through the peering connection.
- On the private instance, configure a cron job to push the `/var/log/boots.log` file to the public instance (using `scp` or `rsync`).
- On the public instance, configure a cron job to push that same file to the created S3 bucket.
- The uploaded file must be stored in the S3 bucket under the path `datacenter-priv-vpc/boot/boots.log`.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)

***Notes:***

Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Create Public VPC and Networking**

Create Public VPC

```
PUB_VPC_ID=$(aws ec2 create-vpc --cidr-block 10.20.0.0/16
--tag-specifications
'ResourceType=vpc,Tags=[{Key=Name,Value=datacenter-pub-vpc}]' --query
Vpc.VpcId --output text)

echo $PUB_VPC_ID
```

Create Public Subnet

```
PUB_SUBNET_ID=$(aws ec2 create-subnet --vpc-id $PUB_VPC_ID --cidr-block
10.20.1.0/24 --tag-specifications
'ResourceType=subnet,Tags=[{Key=Name,Value=datacenter-pub-subnet}]' --query
Subnet.SubnetId --output text)

echo $PUB_SUBNET_ID
```

Internet Gateway & Attachment in Public Subnet

```
IGW_ID=$(aws ec2 create-internet-gateway --tag-specifications
'ResourceType=internet-gateway,Tags=[{Key=Name,Value=datacenter-pub-igw}]'
--query InternetGateway.InternetGatewayId --output text)
echo $IGW_ID
```

Attachment

```
aws ec2 attach-internet-gateway --vpc-id $PUB_VPC_ID --internet-gateway-id
$IGW_ID
```

Route Table & Public Route

```
PUB_RT_ID=$(aws ec2 create-route-table --vpc-id $PUB_VPC_ID
--tag-specifications
'ResourceType=route-table,Tags=[{Key=Name,Value=datacenter-pub-rt}]'
--query RouteTable.RouteTableId --output text)

echo $PUB_RT_ID
```

Create Route

```
aws ec2 create-route --route-table-id $PUB_RT_ID --destination-cidr-block
0.0.0.0/0 --gateway-id $IGW_ID
```

Attach Route Table to the subnet

```
aws ec2 associate-route-table --route-table-id $PUB_RT_ID --subnet-id
$PUB_SUBNET_ID
```

**Create S3 Bucket**

```
aws s3api create-bucket --bucket datacenter-s3-logs-17209 --region
us-east-1
```

**Create IAM Role & Policy**

Create the file

```
touch trust-policy.json
chmod +x trust-policy.json
```

Add the following content in the file

```
{
  "Version": "2012-10-17",
  "Statement": [{ "Effect": "Allow", "Principal": { "Service":
"ec2.amazonaws.com" }, "Action": "sts:AssumeRole" }]
}
```

Create Role

```
aws iam create-role --role-name datacenter-s3-role
```

```
--assume-role-policy-document file://trust-policy.json
```

```
aws iam put-role-policy --role-name datacenter-s3-role --policy-name
S3PutPolicy --policy-document '{
    "Version": "2012-10-17",
    "Statement": [{ "Effect": "Allow", "Action": "s3:PutObject",
"Resource": "arn:aws:s3:::datacenter-s3-logs-17209/*" }]
}'
```

**EC2 Instance**

**Retrieve the Latest AMI ID using SSM Parameter Store**

```
UBUNTU_AMI_ID=$(aws ssm get-parameter --region us-east-1 --name
/aws/service/canonical/ubuntu/server/noble/stable/current/amd64/hvm/ebs-gp3
/ami-id --query 'Parameter.Value' --output text)

echo "Using AMI ID: $UBUNTU_AMI_ID"
```

**Create the security group**

```
PUB_SG_ID=$(aws ec2 create-security-group --group-name datacenter-pub-sg
--description "Security group for Public Instance" --vpc-id $PUB_VPC_ID
--region us-east-1 --query 'GroupId' --output text)

echo "Created Security Group ID: $PUB_SG_ID"
```

**Add HTTP inbound rule (port 80):**

```
aws ec2 authorize-security-group-ingress --group-id $PUB_SG_ID --protocol
tcp --port 80 --cidr 0.0.0.0/0 --region us-east-1
```

**Add HTTP inbound rule (port 22):**

```
aws ec2 authorize-security-group-ingress --group-id $PUB_SG_ID --protocol
tcp --port 22 --cidr 0.0.0.0/0 --region us-east-1
```

**Launch the EC2 Instance**

```
PUB_INSTANCE_ID=$(aws ec2 run-instances --region us-east-1 --image-id
$UBUNTU_AMI_ID --instance-type t2.micro --key-name datacenter-key
--subnet-id $PUB_SUBNET_ID --security-group-ids $PUB_SG_ID --count 1
--tag-specifications
```

```
'ResourceType=instance,Tags=[{Key=Name,Value=datacenter-pub-ec2}]' --query
'Instances[0].InstanceId' --output text)

echo "Launched Instance ID: $PUB_INSTANCE_ID"
```

**Attached the role (datacenter-s3-logs) in to the Instance after the creation.**
**AWS does not always assign a public IP by default. Once an instance is launched without a public IP, We cannot add a standard "auto-assigned" public IP to it later.**

Allocate the Elastic IP

```
EIP_ALLOC=$(aws ec2 allocate-address --domain vpc --query 'AllocationId'
--output text)

echo "New Allocation ID: $EIP_ALLOC"
```

Associate the EIP

```
aws ec2 associate-address --instance-id $PUB_INSTANCE_ID --allocation-id
$EIP_ALLOC
```

**VPC Peering and Routing**

Establish Peering

Get the Private VPC ID

```
PRIV_VPC_ID=$(aws ec2 describe-vpcs --filters
"Name=tag:Name,Values=datacenter-priv-vpc" --query 'Vpcs[0].VpcId' --output
text)
```

Create VPC Peering

```
PEER_ID=$(aws ec2 create-vpc-peering-connection --vpc-id $PRIV_VPC_ID
--peer-vpc-id $PUB_VPC_ID --tag-specifications
'ResourceType=vpc-peering-connection,Tags=[{Key=Name,Value=datacenter-vpc-p
eering}]' --query VpcPeeringConnection.VpcPeeringConnectionId --output
text)

echo $PEER_ID
```

Acceptance

```
aws ec2 accept-vpc-peering-connection --vpc-peering-connection-id $PEER_ID
```

Update Route Tables

Get Private Route Table ID

```
PRIV_RT_ID=$(aws ec2 describe-route-tables --filters
"Name=tag:Name,Values=datacenter-priv-rt" --query
'RouteTables[0].RouteTableId' --output text)
```

Route Private -> Public

```
aws ec2 create-route --route-table-id $PRIV_RT_ID --destination-cidr-block
10.20.0.0/16 --vpc-peering-connection-id $PEER_ID
```

Route Public -> Private

```
aws ec2 create-route --route-table-id $PUB_RT_ID --destination-cidr-block
10.10.0.0/16 --vpc-peering-connection-id $PEER_ID
```

**Log Transfer Automation (Cron Jobs)**

**On Public Instance**

SSH into `datacenter-pub-ec2` from `aws-client` host using it's public ip.

```
ssh -i ~/.ssh/datacenter-key.pem ubuntu@<datacenter-pub-ec2-public-ip>
```

Public Instance → S3 Bucket

Run the Following command in the public instance

```
# Update system
sudo apt-get update
sudo apt-get upgrade -y

# Install required packages
sudo apt install -y unzip curl

## Install aws CLI
# Download Installer
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o
"awscliv2.zip"

# Unzip
```

```
unzip awscliv2.zip
# Run Install program
sudo ./aws/install

# Configure AWS
aws configure

# Enter the Access Key and Secret Key, Get the keys on the Client Host at
~/.aws/credentials

# Add cron job
crontab -e

# Add the below cron job
* * * * * aws s3 cp /home/ubuntu/boot/boots.log
s3://datacenter-s3-logs-17209/datacenter-priv-vpc/boot/boots.log
```

Note: Secure Copy the **datacenter-key.pem** file from client host to the Public Instance at **/home/ubuntu/.ssh/**

SSH into Private Instance from the Public Instance via Private IP of the Private Instance

```
ssh -i /home/ubuntu/.ssh/datacenter-key.pem ubuntu@<PRIVATE_IP>
```

**On Private Instance**
Note: Secure Copy the **datacenter-key.pem** file from client host to the Private Instance at **/home/ubuntu/.ssh/**

Private Instance (ubuntu) → Public Instance

On Private Instance (datacenter-priv-ec2), Add to crontab -e

```
* * * * * scp -i /home/ubuntu/.ssh/datacenter-key.pem /var/log/boots.log
ubuntu@<PUB_INST_PRIVATE_IP>:/home/ubuntu/boot/boots.log
```

# Day 50: Expanding EC2 Instance Storage for Development Needs

## Task Description:

The Nautilus DevOps Team has recently been informed by the Development Team that their EC2 instance is running out of storage space. This instance, crucial for development activities, is named `nautilus-ec2` and currently has an attached volume of `8 GiB`. To accommodate the increasing data requirements, the storage needs to be expanded to `12 GiB`. This change should ensure that the expanded space is immediately available for use within the instance without disrupting ongoing activities.

1. **Identify Volume:** Find the volume attached to the `nautilus-ec2` instance.
2. **Expand Volume:** Increase the volume size from `8 GiB` to `12 GiB`.
3. **Reflect Changes:** Ensure the root (`/`) partition within the instance reflects the expanded size from `8 GiB` to `12 GiB`.
4. **SSH Access:** Use the key pair located at `/root/nautilus-keypair.pem` on the `aws-client` host to SSH into the EC2 instance.

**AWS Credentials:** (You can run the `showcreds` command on `aws-client` host to retrieve these credentials)
***Notes:***
Create the resources only in **us-east-1** region.

## Solution:

First we need to check that terminal configured with AWS or not

```
aws configure list
```

Check IAM user identity

```
aws sts get-caller-identity
```

Check AWS Credentials

```
cat ~/.aws/credentials
```

Check config file

```
cat ~/.aws/config
```

Using AWS CLI

**Identify and Expand the EBS Volume**

Get the Instance ID

```
INSTANCE_ID=$(aws ec2 describe-instances --filters
"Name=tag:Name,Values=nautilus-ec2" --query
'Reservations[0].Instances[0].InstanceId' --output text --region us-east-1)
```

Get the Volume ID

```
VOLUME_ID=$(aws ec2 describe-volumes --filters
"Name=attachment.instance-id,Values=$INSTANCE_ID" --query
'Volumes[0].VolumeId' --output text --region us-east-1)
```

Modify the Volume to 12 GiB

```
aws ec2 modify-volume --volume-id $VOLUME_ID --size 12 --region us-east-1
```

**Extend the Partition and File System**
SSH into the instance to make the OS aware of the additional 4 GiB.

SSH into the instance

```
# Get the Public IP
PUB_IP=$(aws ec2 describe-instances --instance-ids $INSTANCE_ID --query
'Reservations[0].Instances[0].PublicIpAddress' --output text --region
us-east-1)

ssh -i /root/nautilus-keypair.pem ubuntu@$PUB_IP
```

Extend the Partition
Inside the instance, check disk layout using `lsblk`.
Grow the Partition

```
# Use growpart to expand partition 1 of the root disk (e.g., /dev/nvme0n1
or /dev/xvda)
sudo growpart /dev/nvme0n1 1
# OR if using older drivers: sudo growpart /dev/xvda 1
```

Extend the File System
Check your file system type with `df -hT`
If XFS (Standard for Amazon Linux)

```
sudo xfs_growfs -d /
```

If EXT4 (Standard for Ubuntu)

```
sudo resize2fs /dev/nvme0n1p1
# OR /dev/xvda1
```

**Verification**
un `df -h` to verify that the root (`/`) partition now shows a total size of **12 GiB**.