Lab 3 :

> Author: Badr TAJINI - DevOps Data for SWE - ESIEE - 2025

Lab: How to Deploy Your Apps

We will work step-by-step on lab exercises to help you understand different orchestration methods for deploying applications, as covered in Chapter 3, "How to Deploy Your Apps." We will explore:

- **Server Orchestration** using Ansible
- **VM Orchestration** using Packer and OpenTofu (an open-source Terraform fork)
- **Container Orchestration** using Docker and Kubernetes

Table of Contents

Prerequisites

Before starting, ensure you have the following:

- **AWS Account**: An AWS account with permissions to create and manage resources.
- **AWS CLI Installed and Configured**: AWS CLI configured with your AWS credentials.
- **Ansible Installed**: Version 2.9 or later.
- **Packer Installed**: To build VM images.
- **OpenTofu Installed**: Install OpenTofu.
- **Docker Desktop Installed**: Ensure Kubernetes is enabled.
- **Kubectl Installed**: The Kubernetes command-line tool.
- **Git Installed**: To clone repositories from GitHub.
- **SSH Key Pair**: To access EC2 instances via SSH.

Part 1: Server Orchestration with Ansible

Step 1: Set Up the Ansible Environment

1.1 Clone the Sample Code Repository

```
cd devops-book/td3/scripts/ansible
```

1.2 Install Required Ansible Collections

```
ansible-galaxy collection install amazon.aws
```

1.3 Configure AWS Credentials for Ansible

Option 1: Use AWS CLI Configuration

```
aws configure
```

Option 2: Set Environment Variables

```
export AWS_ACCESS_KEY_ID=your_access_key_id
export AWS_SECRET_ACCESS_KEY=your_secret_access_key
export AWS_DEFAULT_REGION=us-east-2
```

Step 2: Creating EC2 Instances with Ansible

2.1 Create a Variables File

Create `sample-app-vars.yml`:

```
num_instances: 3
base_name: sample_app_instances
http_port: 8080
```

2.2 Run the Ansible Playbook to Create EC2 Instances

```
ansible-playbook -v create_ec2_instances_playbook.yml --extra-vars "@sample-app-vars.yml"
```

2.3 Verify the EC2 Instances

- Navigate to the **EC2 Dashboard** in AWS Management Console.
- Confirm three instances are running with the tag `Ansible=sample_app_instances`.

Step 3: Configuring Dynamic Inventory

3.1 Create Inventory Configuration

Create `inventory.aws_ec2.yml`:

```yaml
plugin: amazon.aws.aws_ec2
regions:
  - us-east-2
keyed_groups:
  - key: tags.Ansible
leading_separator: ''
```

3.2 Create Group Variables

Create `group_vars/sample_app_instances.yml`:

```yaml
ansible_user: ec2-user
ansible_ssh_private_key_file: ansible-ch3.key
ansible_host_key_checking: false
```

Step 4: Deploying the Sample Node.js Application

4.1 Create the Application Deployment Playbook

Create `configure_sample_app_playbook.yml`:

```yaml
- name: Configure servers to run the sample-app
  hosts: sample_app_instances
  gather_facts: true
  become: true
  roles:
    - role: nodejs-app
    - role: sample-app
      become_user: app-user
```

4.2 Create the `nodejs-app` Role

Create `roles/nodejs-app/tasks/main.yml`:

```yaml
- name: Add Node.js packages to yum
  shell: curl -fsSL https://rpm.nodesource.com/setup_21.x | bash -

- name: Install Node.js
  yum:
    name: nodejs

- name: Create app user
  user:
    name: app-user

- name: Install PM2
  npm:
    name: pm2
    global: true

- name: Configure PM2 to run at startup as the app user
  shell: |
    su - app-user -c "pm2 startup systemd -u app-user --hp /home/app-user"
    systemctl enable pm2-app-user
```

4.3 Create the `sample-app` Role

Copy `app.js` and `app.config.js` into `roles/sample-app/files/`.

- **`app.js`**:

  ```javascript
  const http = require('http');
  const server = http.createServer((req, res) => {
    res.end('Hello, World!\n');
  });
  server.listen(8080, () => {
    console.log('Listening on port 8080');
  });
  ```

- **`app.config.js`**:

```
module.exports = {
  apps : [{
    name   : "sample-app",
    script : "./app.js",
    exec_mode: "cluster",
    instances: "max",
    env: {
      "NODE_ENV": "production"
    }
  }]
}
```

Create `roles/sample-app/tasks/main.yml`:

```
- name: Copy sample app
  copy:
    src: "{{ item }}"
    dest: "/home/app-user/"
    owner: app-user
    group: app-user
    mode: 0644
  with_fileglob:
    - "files/*"

- name: Start sample app using PM2
  shell: pm2 start app.config.js
  args:
    chdir: /home/app-user/
  become_user: app-user

- name: Save PM2 process list
  shell: pm2 save
  become_user: app-user
```

4.4 Run the Application Deployment Playbook

```
ansible-playbook -v -i inventory.aws_ec2.yml configure_sample_app_playbook.yml
```

4.5 Verify the Application is Running

Get public IPs:

```
aws ec2 describe-instances \
  --filters "Name=tag:Ansible,Values=sample_app_instances" \
  --query "Reservations[*].Instances[*].PublicIpAddress" \
  --output text
```

Test the application:

```
curl http://<EC2_INSTANCE_PUBLIC_IP>:8080
```

Step 5: Setting Up Nginx as a Load Balancer

5.1 Create a Variables File for Nginx

Create `nginx-vars.yml` :

```yaml
num_instances: 1
base_name: nginx_instances
http_port: 80
```

5.2 Run the Playbook to Create an EC2 Instance for Nginx

```
ansible-playbook -v create_ec2_instances_playbook.yml --extra-vars "@nginx-vars.yml"
```

5.3 Create Group Variables for Nginx

Create `group_vars/nginx_instances.yml` :

```yaml
ansible_user: ec2-user
ansible_ssh_private_key_file: ansible-ch3.key
ansible_host_key_checking: false
```

5.4 Create the Nginx Playbook

Create `configure_nginx_playbook.yml` :

```yaml
- name: Configure servers to run Nginx
  hosts: nginx_instances
  gather_facts: true
  become: true
  roles:
    - role: nginx
```

5.5 Create the `nginx` Role

Create `roles/nginx/templates/nginx.conf.j2`:

```
user nginx;
worker_processes auto;
error_log /var/log/nginx/error.log notice;
pid /run/nginx.pid;

events {
    worker_connections 1024;
}

http {
    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;

    include             /etc/nginx/mime.types;
    default_type        application/octet-stream;

    upstream backend {
        {% for host in groups['sample_app_instances'] %}
        server {{ hostvars[host]['ansible_host'] }}:8080;
        {% endfor %}
    }

    server {
        listen       80;
        listen       [::]:80;

        location / {
            proxy_pass http://backend;
        }
    }
}
```

Create `roles/nginx/tasks/main.yml`:

```yaml
- name: Install Nginx
  yum:
    name: nginx
    state: present

- name: Copy Nginx config
  template:
    src: nginx.conf.j2
    dest: /etc/nginx/nginx.conf
  notify:
    - restart nginx

- name: Start and enable Nginx
  service:
    name: nginx
    state: started
    enabled: true

handlers:
  - name: restart nginx
    service:
      name: nginx
      state: restarted
```

5.6 Run the Nginx Playbook

```
ansible-playbook -v -i inventory.aws_ec2.yml configure_nginx_playbook.yml
```

5.7 Verify the Load Balancer

Get Nginx public IP:

```
aws ec2 describe-instances \
  --filters "Name=tag:Ansible,Values=nginx_instances" \
  --query "Reservations[*].Instances[*].PublicIpAddress" \
  --output text
```

Test the load balancer:

```
curl http://<NGINX_PUBLIC_IP>
```

Step 6: Implementing Rolling Updates

6.1 Enable Rolling Updates in the Playbook

Modify `configure_sample_app_playbook.yml`:

```
- name: Configure servers to run the sample-app
  hosts: sample_app_instances
  gather_facts: true
  become: true
  serial: 1
  max_fail_percentage: 30
  roles:
    - role: nodejs-app
    - role: sample-app
      become_user: app-user
```

6.2 Update the Application

Modify `roles/sample-app/files/app.js`:

```
res.end('DevOps Base!\n');
```

6.3 Run the Application Deployment Playbook Again

```
ansible-playbook -v -i inventory.aws_ec2.yml configure_sample_app_playbook.yml
```

6.4 Verify the Rolling Update

Continuously test the application:

```
while true; do curl http://<NGINX_PUBLIC_IP>; sleep 1; done
```

Part 2: VM Orchestration with Packer and OpenTofu

Step 1: Building a VM Image Using Packer

1.1 Set Up the Working Directory

```
mkdir -p devops_base/td3/scripts/packer
cd devops_base/td3/scripts/packer
```

1.2 Create the Packer Template

Create `sample-app.pkr.hcl`:

```hcl
packer {
  required_plugins {
    amazon = {
      version = ">= 1.0.0"
      source  = "github.com/hashicorp/amazon"
    }
  }
}

variable "aws_region" {
  type    = string
  default = "us-east-2"
}

source "amazon-ebs" "amazon_linux" {
  ami_name      = "packer-sample-app-{{timestamp}}"
  instance_type = "t2.micro"
  region        = var.aws_region
  source_ami_filter {
    filters = {
      name                = "amzn2-ami-hvm-*-x86_64-gp2"
      root-device-type    = "ebs"
      virtualization-type = "hvm"
    }
    owners      = ["amazon"]
    most_recent = true
  }
  ssh_username = "ec2-user"
}

build {
  sources = ["source.amazon-ebs.amazon_linux"]

  provisioner "file" {
    sources     = ["app.js", "app.config.js"]
    destination = "/tmp/"
  }

  provisioner "shell" {
    inline = [
      "sudo yum update -y",
      "curl -fsSL https://rpm.nodesource.com/setup_21.x | sudo bash -",
      "sudo yum install -y nodejs",
      "sudo useradd app-user",
      "sudo mkdir -p /home/app-user",
      "sudo mv /tmp/app.js /tmp/app.config.js /home/app-user/",
      "sudo chown -R app-user:app-user /home/app-user",
      "sudo npm install pm2@latest -g",
      "sudo su - app-user -c 'pm2 startup systemd -u app-user --hp /home/app-user'",
      "sudo systemctl enable pm2-app-user",
```

```
    ]
  }
}
```

1.3 Prepare the Application Files

Create `app.js`:

```
const http = require('http');
const server = http.createServer((req, res) => {
  res.end('Hello, World!\n');
});
server.listen(8080, () => {
  console.log('Listening on port 8080');
});
```

Create `app.config.js`:

```
module.exports = {
  apps : [{
    name        : "sample-app",
    script      : "./app.js",
    exec_mode   : "cluster",
    instances   : "max",
    env: {
      "NODE_ENV": "production"
    }
  }]
}
```

1.4 Initialize and Build the Packer Image

Initialize Packer:

```
packer init sample-app.pkr.hcl
```

Build the image:

```
packer build sample-app.pkr.hcl
```

Step 2: Deploying the VM Image Using OpenTofu

2.1 Set Up the OpenTofu Working Directory

```
mkdir -p ../tofu/live/asg-sample
cd ../tofu/live/asg-sample
```

2.2 Create the Main OpenTofu Configuration

Create `main.tf`:

```
provider "aws" {
  region = "us-east-2"
}

module "asg" {
  source = "github.com/your_github_name/devops-base//td3/tofu/modules/asg"

  name          = "sample-app-asg"
  ami_id        = "<YOUR_AMI_ID>"  # Replace with your AMI ID
  user_data     = filebase64("${path.module}/user-data.sh")
  app_http_port = 8080

  instance_type    = "t2.micro"
  min_size         = 1
  max_size         = 10
  desired_capacity = 3
}
```

2.3 Create the User Data Script

Create `user-data.sh`:

```
#!/usr/bin/env bash

set -e

sudo su - app-user -c "
  pm2 start /home/app-user/app.config.js && \
  pm2 save
"
```

2.4 Initialize and Apply OpenTofu Configuration

Initialize OpenTofu:

```
tofu init
```

Apply the configuration:

```
tofu apply
```

Step 3: Deploying an Application Load Balancer (ALB)

3.1 Update the OpenTofu Configuration to Include the ALB

Modify `main.tf`:

```
module "alb" {
  source = "github.com/your_github_name/devops-base//td3/tofu/modules/alb"

  name                  = "sample-app-alb"
  alb_http_port         = 80
  app_http_port         = 8080
  app_health_check_path = "/"
}

module "asg" {
  # ... existing configuration ...

  target_group_arns = [module.alb.target_group_arn]
}
```

3.2 Add Output for ALB DNS Name

Create `outputs.tf`:

```
output "alb_dns_name" {
  description = "The ALB's domain name"
  value       = module.alb.alb_dns_name
}
```

3.3 Apply the Updated Configuration

```
tofu apply
```

3.4 Test the Application Through the ALB

Retrieve the ALB DNS name:

```
tofu output alb_dns_name
```

Test the application:

```
curl http://<ALB_DNS_NAME>
```

Step 4: Implementing Rolling Updates with ASG Instance Refresh

4.1 Enable Instance Refresh in the ASG Configuration

Update `main.tf`:

```
module "asg" {
  # ... existing configuration ...

  instance_refresh = {
    min_healthy_percentage = 100
    max_batch_size         = 1
    strategy               = "Rolling"
    auto_rollback          = true
  }
}
```

4.2 Apply the Configuration

```
tofu apply
```

4.3 Update the Application Code

Modify `app.js` in the Packer directory:

```
res.end('DevOps Base!\n');
```

4.4 Rebuild the Packer Image

```
cd ../../packer
packer build sample-app.pkr.hcl
```

4.5 Update the AMI ID in OpenTofu Configuration

Update `ami_id` in `main.tf` with the new AMI ID.

4.6 Apply the Configuration to Trigger Rolling Update

```
cd ../tofu/live/asg-sample
tofu apply
```

4.7 Verify Zero-Downtime Deployment

Monitor the application:

```
while true; do curl http://<ALB_DNS_NAME>; sleep 1; done
```

Part 3: Container Orchestration with Docker and Kubernetes

Step 1: Building and Running the Docker Image Locally

1.1 Set Up the Working Directory

```
mkdir -p devops_base/td3/scripts/docker
cd devops_base/td3/scripts/docker
```

1.2 Create the Sample Application

Create `app.js`:

```
const http = require('http');
const server = http.createServer((req, res) => {
  res.end('Hello, World!\n');
});
server.listen(8080, () => {
  console.log('Listening on port 8080');
});
```

1.3 Create the Dockerfile

Create `Dockerfile`:

```
FROM node:current-alpine

WORKDIR /usr/src/app

COPY app.js .

EXPOSE 8080

CMD ["node", "app.js"]
```

1.4 Build the Docker Image

```
docker build -t sample-app:v1 .
```

1.5 Run the Docker Container Locally

```
docker run -p 8080:8080 --name sample-app --rm sample-app:v1
```

1.6 Test the Application

In a separate terminal:

```
curl http://localhost:8080
```

Step 2: Deploying the Application to a Local Kubernetes Cluster

2.1 Enable Kubernetes in Docker Desktop

- Open Docker Desktop.
- Go to **Settings** > **Kubernetes**.
- Check **Enable Kubernetes**.
- Click **Apply & Restart**.

2.2 Verify Kubernetes is Running

```
kubectl get nodes
```

2.3 Create a Kubernetes Deployment Configuration

Create `sample-app-deployment.yaml`:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-app-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
      - name: sample-app
        image: sample-app:v1
        ports:
        - containerPort: 8080
```

2.4 Apply the Deployment Configuration

```
kubectl apply -f sample-app-deployment.yaml
```

2.5 Verify the Pods are Running

```
kubectl get pods
```

2.6 Create a Kubernetes Service Configuration

Create `sample-app-service.yaml`:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: sample-app-service
spec:
  type: LoadBalancer
  selector:
    app: sample-app
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

2.7 Apply the Service Configuration

```
kubectl apply -f sample-app-service.yaml
```

2.8 Test the Application Through the Service

```bash
curl http://localhost
```

Step 3: Performing a Rolling Update

3.1 Update the Application Code

Modify `app.js`:

```javascript
res.end('DevOps Base!\n');
```

3.2 Build a New Docker Image

```bash
docker build -t sample-app:v2 .
```

3.3 Update the Deployment to Use the New Image

Edit `sample-app-deployment.yaml`:

```yaml
...
      - name: sample-app
        image: sample-app:v2
...
```

3.4 Apply the Updated Deployment Configuration

```
kubectl apply -f sample-app-deployment.yaml
```

3.5 Monitor the Rolling Update

```
kubectl rollout status deployment/sample-app-deployment
```

3.6 Test the Updated Application

```bash
curl http://localhost
```

Important Note for the sections about EKS (Step 4-5-6-7)

Warning: EKS is *not* part of the AWS free tier. Running the examples in this section will incur charges. As of June 2024, the pricing is $0.10 per hour for the control plane. Ensure you clean up all resources after completing the lab to avoid unnecessary costs.

(You could go to the next section : Part 4: Deploying Applications Using Serverless Orchestration with AWS Lambda)

Step 4: Deploying a Kubernetes Cluster in AWS Using EKS

4.1 Set Up the Working Directory

Create a directory for the EKS cluster configuration:

```bash
mkdir -p devops_base/td3/scripts/tofu/live/eks-sample
cd devops_base/td3/scripts/tofu/live/eks-sample
```

4.2 Configure the `eks-cluster` Module

Create a file named `main.tf` with the following content:

Example 3-29: Configure the `eks-cluster` module (`ch3/tofu/live/eks-sample/main.tf`)

```
provider "aws" {
  region = "us-east-2"
}

module "cluster" {
  source = "github.com/your_github_name/devops-base//td3/tofu/modules/eks-cluster"

  name        = "eks-sample"          # (1)
  eks_version = "1.29"                # (2)

  instance_type        = "t2.micro"  # (3)
  min_worker_nodes     = 1           # (4)
  max_worker_nodes     = 10          # (5)
  desired_worker_nodes = 3           # (6)
}
```

This code configures the following parameters:

1. `name` : The name to use for the control plane, worker nodes, and all other resources created by the module.
2. `eks_version` : The version of Kubernetes to use (e.g., "1.29").
3. `instance_type` : The type of EC2 instance to use for worker nodes.
4. `min_worker_nodes` : The minimum number of worker nodes to run.
5. `max_worker_nodes` : The maximum number of worker nodes to run.
6. `desired_worker_nodes` : The initial number of worker nodes to run.

4.3 Deploy the EKS Cluster

Initialize OpenTofu:

```
tofu init
```

Apply the configuration:

```
tofu apply
```

Type `yes` when prompted to confirm the creation of resources.

Note: Deployment of the EKS cluster may take several minutes (typically around 10-15 minutes).

4.4 Configure `kubectl` to Connect to the EKS Cluster

Once the EKS cluster is deployed, you need to configure `kubectl` to communicate with it.

Run the following command:

```
aws eks update-kubeconfig --region us-east-2 --name eks-sample
```

This command updates your local `kubeconfig` file with the cluster information.

4.5 Verify the EKS Cluster

Check the nodes in the cluster:

```
kubectl get nodes
```

You should see output similar to:

```
NAME                                          STATUS   ROLES    AGE   VERSION
ip-192-168-xx-xx.us-east-2.compute.internal   Ready    <none>   5m    v1.29.x
ip-192-168-xx-xx.us-east-2.compute.internal   Ready    <none>   5m    v1.29.x
ip-192-168-xx-xx.us-east-2.compute.internal   Ready    <none>   5m    v1.29.x
```

This output indicates that your cluster has three worker nodes running and ready.

Step 5: Pushing a Docker Image to Amazon ECR

5.1 Build the Docker Image for the Sample Application

Assuming you have a Dockerfile for your sample application, navigate to the directory containing the Dockerfile (e.g., `devops_base/td3/scripts/docker`):

```
cd ../../../docker
```

If you don't have the Dockerfile, create it with the following content:

```
# Dockerfile

FROM node:current-alpine

WORKDIR /usr/src/app

COPY app.js .

EXPOSE 8080

CMD ["node", "app.js"]
```

Also, ensure you have `app.js` with the following content:

```
// app.js

const http = require('http');
const server = http.createServer((req, res) => {
  res.end('DevOps Base!\n');
});
server.listen(8080, () => {
  console.log('Listening on port 8080');
});
```

Build the Docker Image

First, create a multi-platform builder if you haven't already:

```
docker buildx create --use --name multi-platform-builder
```

Build the Docker image for both `linux/amd64` and `linux/arm64` platforms:

```
docker buildx build \
  --platform=linux/amd64,linux/arm64 \
  --load \
  -t sample-app:v3 \
  .
```

5.2 Create an ECR Repository

Create a new directory for the ECR module:

```
mkdir -p ../../tofu/live/ecr-sample
cd ../../tofu/live/ecr-sample
```

Create a file named `main.tf` with the following content:

Example 3-30: Configure the `ecr-repo` module (`ch3/tofu/live/ecr-sample/main.tf`)

```
provider "aws" {
  region = "us-east-2"
}

module "repo" {
  source = "github.com/your_github_name/devops-base//td3/tofu/modules/ecr-repo"

  name = "sample-app"
}
```

This code will create an ECR repository called `sample-app`.

Create an `outputs.tf` file to output the registry URL:

Example 3-31: Define output variables (`ch3/tofu/live/ecr-sample/outputs.tf`)

```
output "registry_url" {
  description = "URL of the ECR repo"
  value       = module.repo.registry_url
}
```

Initialize OpenTofu:

```
tofu init
```

Apply the configuration:

```
tofu apply
```

After completion, you should see an output similar to:

```
Outputs:

registry_url = "111122223333.dkr.ecr.us-east-2.amazonaws.com/sample-app"
```

Make a note of the `registry_url`.

Step 6: Tag and Push the Docker Image to ECR

6.1 Tag the Docker Image

Replace `<YOUR_ECR_REPO_URL>` with the actual `registry_url` from the previous step.

```
docker tag sample-app:v3 <YOUR_ECR_REPO_URL>:v3
```

Step 6.2 Authenticate Docker to ECR

Run the following command to authenticate Docker to your ECR registry:

```
aws ecr get-login-password --region us-east-2 | \
docker login --username AWS --password-stdin <YOUR_ECR_REPO_URL>
```

6.3 Push the Docker Image to ECR

Push the tagged image to ECR:

```
docker push <YOUR_ECR_REPO_URL>:v3
```

This process may take a few minutes.

Step 7: Deploying the Sample Application to the EKS Cluster

7.1 Update the Kubernetes Deployment YAML

Go back to the directory containing your Kubernetes manifests (e.g., `devops_base/td3/scripts/kubernetes`):

```
cd ../../kubernetes
```

Edit the `sample-app-deployment.yml` file.

Example 3-32: Update the Deployment to use the Docker image from your ECR repo (`ch3/kubernetes/sample-app-deployment.yml`)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-app-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
        - name: sample-app
          image: <YOUR_ECR_REPO_URL>:v3  # Update with your ECR image
          ports:
            - containerPort: 8080
```

Replace `<YOUR_ECR_REPO_URL>` with the actual `registry_url` from earlier.

7.2 Apply the Kubernetes Manifests

Ensure your `kubectl` context is set to your EKS cluster.

Apply the deployment and service manifests:

```
kubectl apply -f sample-app-deployment.yml
kubectl apply -f sample-app-service.yml
```

7.3 Verify the Deployment

Check the status of the pods:

```
kubectl get pods
```

You should see output similar to:

```
NAME                                 READY   STATUS    RESTARTS   AGE
sample-app-deployment-xxxxxx-xxxxx   1/1     Running   0          1m
sample-app-deployment-xxxxxx-xxxxx   1/1     Running   0          1m
sample-app-deployment-xxxxxx-xxxxx   1/1     Running   0          1m
```

Check the services:

```
kubectl get services
```

You should see:

```
NAME                      TYPE           CLUSTER-IP       EXTERNAL-IP
PORT(S)        AGE
kubernetes                ClusterIP      10.xxx.xxx.xxx   <none>
443/TCP        1h
sample-app-loadbalancer   LoadBalancer   10.xxx.xxx.xxx   abcdef1234.us-east-
2.elb.amazonaws.com   80:xxxx/TCP   1m
```

7.4 Test the Application

Retrieve the `EXTERNAL-IP` of the `sample-app-loadbalancer` service.

Use `curl` or a web browser to access the application:

```
curl http://abcdef1234.us-east-2.elb.amazonaws.com
```

You should see:

```
DevOps Base!
```

Note: It may take a few minutes for the Load Balancer to become available and for the health checks to pass. If you receive connection errors, wait a few minutes and try again.

Exercice [Practice]

Here are some exercises you can try to deepen your understanding:

- **Deploy an Application Load Balancer (ALB)**:

By default, if you deploy a Kubernetes Service of type `LoadBalancer` into EKS, EKS will create a *Classic Load Balancer*. To deploy an ALB instead, you need to:

- Install the AWS Load Balancer Controller.
- Annotate your service manifest accordingly.

Refer to the AWS documentation for detailed steps.

- **Scale the Deployment**:

  Modify the `replicas` field in your deployment manifest to scale the number of pods up or down.

- **Test Fault Tolerance**:

  Terminate one of the worker node instances using the AWS Console. Observe how the cluster recovers and maintains application availability.

Cleanup

To avoid incurring charges, destroy the resources when you're done.

Step 1: Delete the Kubernetes Resources

```
kubectl delete -f sample-app-deployment.yml
kubectl delete -f sample-app-service.yml
```

Step 2: Destroy the EKS Cluster

Navigate to the `eks-sample` directory:

```
cd ../tofu/live/eks-sample
```

Destroy the resources:

```
tofu destroy
```

Type `yes` when prompted.

Step 3: Destroy the ECR Repository

Navigate to the `ecr-sample` directory:

```
cd ../ecr-sample
```

Destroy the resources:

```
tofu destroy
```

Type `yes` when prompted.

Step 4: Delete the Docker Images from ECR

If any images remain in ECR, you can delete them via the AWS Console or using the AWS CLI:

```
aws ecr batch-delete-image --repository-name sample-app --image-ids imageTag=v3
```

Conclusion

By completing this lab, you've:

- Deployed a Kubernetes cluster in AWS using Amazon EKS.
- Pushed a Docker image to Amazon ECR.
- Deployed a Dockerized application to the EKS cluster.
- Practiced working with Kubernetes in a cloud environment.
- Understood the steps involved in container orchestration using Kubernetes on AWS.

This hands-on experience demonstrates how to manage containerized applications using Kubernetes in a production-like environment.

Note: Always ensure you manage AWS resources responsibly to prevent unnecessary costs.

Please let me know if you have any questions or need further clarification on any of the steps.

Conclusion

By completing these labs, you've:

- Explored **server orchestration** with Ansible.
- Practiced **VM orchestration** using Packer and OpenTofu.
- Delved into **container orchestration** with Docker and Kubernetes.
- Implemented rolling updates and zero-downtime deployments.
- Learned how to manage infrastructure and applications efficiently across different orchestration methods.

Part 4: Deploying Applications Using Serverless Orchestration with AWS Lambda

In this part of our lab, we will:

- **Create a serverless function** using AWS Lambda.
- **Deploy the function** and configure it to respond to HTTP requests via API Gateway.
- **Update the function** to implement changes quickly.
- **Understand the benefits and limitations** of serverless orchestration.

Prerequisites

Before starting, ensure you have the following:

- **AWS Account**: An AWS account with permissions to create and manage resources.
- **AWS CLI Installed and Configured**: AWS CLI configured with your AWS credentials.
- **OpenTofu Installed**: Install OpenTofu.
- **Git Installed**: To clone repositories from GitHub.
- **Node.js Installed**: For local development.

Part 4: Serverless Orchestration with AWS Lambda

Step 1: Set Up the Working Directory

Create a directory for the Lambda function:

```
mkdir -p devops_base/td3/scripts/tofu/live/lambda-sample/src
cd devops_base/td3/scripts/tofu/live/lambda-sample
```

Step 2: Create the Lambda Function Code

Inside the `src` directory, create a file named `index.js` with the following content:

Example 3-34: The handler code in `index.js`

```
exports.handler = (event, context, callback) => {
  callback(null, { statusCode: 200, body: "Hello, World!" });
};
```

This simple Lambda function:

- Exports a handler function that AWS Lambda can invoke.
- Returns a 200 OK response with the body "Hello, World!".

Step 3: Create the Main OpenTofu Configuration

Create a file named `main.tf` in the `lambda-sample` directory with the following content:

Example 3-33: Configure the `lambda` module

```
provider "aws" {
  region = "us-east-2"
}


module "function" {
  source = "github.com/your_github_name/devops-base//td3/tofu/modules/lambda"

  name     = "lambda-sample"         # (1)

  src_dir = "${path.module}/src"     # (2)
  runtime = "nodejs20.x"             # (3)
  handler = "index.handler"          # (4)

  memory_size = 128                  # (5)
  timeout     = 5                    # (6)

  environment_variables = {          # (7)
    NODE_ENV = "production"
  }

  # ... (other params omitted) ...
}
```

This code sets the following parameters:

1. `name`: The name to use for the Lambda function and all other resources created by this module.
2. `src_dir`: The directory which contains the code for the Lambda function (`src` folder).
3. `runtime`: The runtime used by this function (`nodejs20.x` for Node.js 20.x).
4. `handler`: The entry point to call your function. The format is `<FILE>.<FUNCTION>`, where `<FILE>` is the file in your deployment package and `<FUNCTION>` is the name of the function to call in that file (`index.handler` in this case).
5. `memory_size`: The amount of memory (in MB) to give the Lambda function.
6. `timeout`: The maximum amount of time (in seconds) the Lambda function has to run.
7. `environment_variables`: Environment variables to set for the function.

Step 4: Deploy the Lambda Function

Initialize OpenTofu:

```
tofu init
```

Apply the configuration:

```
tofu apply
```

Type `yes` when prompted to confirm the creation of resources.

Step 5: Verify the Lambda Function

1. **Open the AWS Lambda Console**:
   - Navigate to the AWS Lambda Console.
   - You should see a function named `lambda-sample`.
2. **Test the Function Manually**:
   - Click on the `lambda-sample` function.
   - Click on the **Test** button.
   - For the test event, you can use the default settings.
   - Click **Test** to invoke the function.
   - You should see the response with status code `200` and body `"Hello, World!"`.

Step 6: Set Up API Gateway to Trigger the Lambda Function

Step 6.1: Update the OpenTofu Configuration

Add the `api-gateway` module to your `main.tf`:

Example 3-35: Configure the `api-gateway` module to trigger the Lambda function

```
module "gateway" {
  source = "github.com/your_github_name/devops-base//td3/tofu/modules/api-gateway"

  name               = "lambda-sample"              # (1)
  function_arn       = module.function.function_arn # (2)
  api_gateway_routes = ["GET /"]                    # (3)
}
```

This code sets the following parameters:

1. `name`: The name to use for the API Gateway and all other resources created by the module.
2. `function_arn`: The Amazon Resource Name (ARN) of the Lambda function the API Gateway should trigger.
3. `api_gateway_routes`: The routes that should trigger the Lambda function (HTTP GET to the `/` path).

Step 6.2: Add the API Endpoint as an Output Variable

Create a file named `outputs.tf` with the following content:

Example 3-36: Add the API Gateway domain name as an output variable

```
output "api_endpoint" {
  description = "The API Gateway endpoint"
  value       = module.gateway.api_endpoint
}
```

Step 7: Deploy the API Gateway Configuration

Apply the updated configuration:

```
tofu apply
```

Type `yes` when prompted.

Step 8: Test the API Endpoint

1. **Retrieve the API Endpoint**:

   After the apply completes, you should see an output similar to:

   ```
   Apply complete! Resources: X added, 0 changed, 0 destroyed.

   Outputs:

   api_endpoint = "https://xxxxxxxxxx.execute-api.us-east-2.amazonaws.com"
   ```

2. **Test the API Endpoint**:

   Use `curl` or a web browser to access the endpoint:

   ```
   curl https://xxxxxxxxxx.execute-api.us-east-2.amazonaws.com
   ```

   Replace `https://xxxxxxxxxx.execute-api.us-east-2.amazonaws.com` with the actual `api_endpoint` value.

   You should receive:

   ```
   Hello, World!
   ```

Step 9: Update the Lambda Function

Step 9.1: Modify the Lambda Function Code

Update the `index.js` file in the `src` directory to change the response:

Example 3-37: Update the response text

```
exports.handler = (event, context, callback) => {
  callback(null, { statusCode: 200, body: "DevOps Base!" });
};
```

Step 9.2: Re-Deploy the Updated Function

Apply the configuration again:

```
tofu apply
```

Step 10: Verify the Update

Test the API endpoint again:

```
curl https://xxxxxxxxxx.execute-api.us-east-2.amazonaws.com
```

You should now see:

```
DevOps Base!
```

Exercice [Practice]

To deepen your understanding, try the following exercises:

- **Experiment with Different Runtimes**: Modify the Lambda function to use a different runtime (e.g., Python, Go) and adjust the code accordingly.
- **Add Additional Routes**: Configure the API Gateway to handle more routes and methods (e.g., `POST /data`).
- **Implement Error Handling**: Update the Lambda function to handle errors and return appropriate HTTP status codes.
- **Integrate with Other AWS Services**: Configure the Lambda function to interact with services like Amazon S3 or DynamoDB.

Cleanup

To avoid incurring charges, destroy the resources when you're done.

Step 1: Destroy OpenTofu Resources

In the `lambda-sample` directory, run:

```
tofu destroy
```

Type `yes` when prompted to confirm the destruction of resources.

Step 2: Verify Resources are Deleted

- **Check AWS Lambda Console**: Ensure the `lambda-sample` function is no longer present.
- **Check API Gateway Console**: Ensure the API created is deleted.

Conclusion

By completing this lab, you've:

- Deployed a serverless function using AWS Lambda.
- Configured API Gateway to trigger the Lambda function in response to HTTP requests.
- Performed rapid updates to your function, demonstrating the speed of serverless deployments.
- Understood the benefits of serverless orchestration, such as focusing on code rather than infrastructure and achieving quick deployment cycles.

This last section of our lab demonstrates **serverless orchestration**, where you deploy and manage functions without having to think about servers at all.

END