PROJECT : Infrastructure automation, CI/CD pipelines, monitoring, and security

Objective: To evaluate your skills in DevOps practices, focusing on infrastructure automation, CI/CD pipelines, monitoring, and security within the AWS environment.

1. **Automated Infrastructure Setup**
   - Use **Terraform modules** to provision a AWS environment with the following components:
     - One **VPC** with two subnets (public and private).
     - A **EC2** instance in the public subnet with a web server installed (e.g., Nginx
     - or Apache).
     - Firewall rules that allow HTTP/HTTPS traffic to the instance.

2. **CI/CD Pipeline**
   - Set up a basic CI/CD pipeline using **Jenkins** to deploy a simple web application.
     - Configure the pipeline to:
       - Pull code from a **Git** repository.
       - Build and test the application.
       - Deploy the application to the EC2 instance.
     - Include automated tests and ensure they pass before deployment.

3. **Monitoring and Logging**
   - Integrate a suitable monitoring tool like AWS **CloudWatch t**o monitor the instance's CPU, memory, and disk usage.
   - Set up a basic alert to notify you via email if CPU usage exceeds 80% using tool like AWS **SNS/SES**

1. Infrastructure setup - Complete Terraform Configuration

Module Structure

The module structure to include separate files for variables and outputs:

```
--------------------------------------------------------------------------------

modules/
├── vpc
│   ├── main.tf
│   ├── variables.tf
│   └── outputs.tf
└── webserver
    ├── main.tf
    ├── variables.tf
    └── outputs.tf
main.tf
--------------------------------------------------------------------------------
```

Main Configuration (main.tf)

```
# Configure the required provider
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

provider "aws" {
  region = "us-east-1" # Replace with your desired region
}

variable "ami" {
  type        = string
  description = "AMI ID for the EC2 instance"
}

variable "instance_type" {
  type        = string
  description = "Instance type for the EC2 instance"
}

module "vpc" {
  source = "./modules/vpc"

  cidr_block                  = "10.0.0.0/16"
  public_subnet_cidr_block    = "10.0.1.0/24"
  private_subnet_cidr_block   = "10.0.2.0/24"
  public_subnet_availability_zone = "us-east-1a"
  private_subnet_availability_zone = "us-east-1b"
}
```

```
module "webserver" {
  source             = "./modules/webserver"
  vpc_id              = module.vpc.vpc_id
  public_subnet_id   = module.vpc.public_subnet_id
  ami                = var.ami
  instance_type      = var.instance_type
}
```

VPC Module

modules/vpc/variables.tf

```
variable "cidr_block" {
  type        = string
  description = "CIDR block for the VPC"
}

variable "public_subnet_cidr_block" {
  type        = string
  description = "CIDR block for the public subnet"
}

variable "private_subnet_cidr_block" {
  type        = string
  description = "CIDR block for the private subnet"
}

variable "public_subnet_availability_zone" {
  type        = string
  description = "Availability zone for the public subnet"
}

variable "private_subnet_availability_zone" {
  type        = string
  description = "Availability zone for the private subnet"
}
```

modules/vpc/outputs.tf

```
output "vpc_id" {
  value = aws_vpc.main.id
}

output "public_subnet_id" {
  value = aws_subnet.public.id
}

output "private_subnet_id" {
  value = aws_subnet.private.id
}
```

modules/vpc/main.tf

```
# Reference variables from variables.tf
data "null_data" "source" {
  count = varset("cidr_block") != null ? 1 : 0
}

resource "aws_vpc" "main" {
  cidr_block = var.cidr_block

  tags = {
    Name = "main-vpc"
  }
}

resource "aws_subnet" "public" {
  vpc_id            = aws_vpc.main.id
  cidr_block       = var.public_subnet_cidr_block
  availability_zone = var.public_subnet_availability_zone

  tags = {
    Name = "public-subnet"
  }
}

resource "aws_subnet" "private" {
  vpc_id            = aws_vpc.main.id
  cidr_block       = var.private_subnet_cidr_block
  availability_zone = var.private_subnet_availability_zone

  tags = {
    Name = "private-subnet"
  }
}

resource "aws_internet_gateway" "main" {
  vpc_id = aws_vpc.main.id
}

resource "aws_route_table" "public" {
  vpc_id = aws_vpc.main.id
}

resource "aws_route"  "internet_route" {
  route_table_id         = aws_route_table.public.id
  destination_cidr_block = "0.0.0.0/0"
  gateway_id             = aws_internet_gateway.main.id
}
```

Web Server Module

modules/webserver/variables.tf

```
variable "vpc_id" {
  type        = string
  description = "ID of the VPC"
}

variable "public_subnet_id" {
  type        = string
  description = "ID of the public subnet"
}

variable  "ami" {
  type        = string
  description = "AMI ID for the EC2 instance"
}

variable "instance_type" {
  type        = string
  description = "Instance type for the EC2 instance"
}
```

modules/webserver/outputs.tf

```
output "public_ip" {
  value = aws_instance.web.public_ip
}
```

modules/webserver/main.tf

```
# Reference variables from variables.tf
resource "aws_security_group" "web" {
  name        = "web-sg"
  description = "Allow HTTP and HTTPS traffic"
  vpc_id      = var.vpc_id

  ingress {
    description = "Allow HTTP traffic"
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr blocks = ["0.0.0.0/0"]
  }

  ingress {
    description  = "Allow HTTPS traffic"
    from port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
```

```
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_instance" "web" {
  ami           = var.ami
  instance_type = var.instance_type
  subnet_id     = var.public_subnet_id
  security_groups = [aws_security_group.web.id]

  tags = {
    Name = "web-server"
  }
}
```

2. CI/CD pipeline using Jenkins to deploy a web application

Setting up a CI/CD pipeline using Jenkins to deploy a web application involves several steps, from configuring Jenkins to automating the build, test, and deployment process. Below is a step-by-step guide:

Step 1: Prerequisites

1. **Jenkins Installation**: Ensure Jenkins is installed and running on a server. You can use an EC2 instance or a local server.
2. **Git Repository**: Have a Git repository containing your web application. This could be hosted on GitHub, GitLab, or any other Git service.
3. **EC2 Instance**: An EC2 instance where you will deploy the web application.
4. **SSH Access**: Ensure that Jenkins has SSH access to the EC2 instance for deployment.
5. **Jenkins Plugins**:
    - **Git Plugin**: To pull code from the repository.
    - **SSH Plugin**: To execute commands on the remote EC2 instance.
    - **Pipeline Plugin**: To define and manage your CI/CD pipelines.

Step 2: Configure Jenkins

1. Install Required Plugins

- Go to `Manage Jenkins` > `Manage Plugins`.
- Under the `Available` tab, search for and install the **Git Plugin**, **SSH Agent Plugin**, and **Pipeline Plugin**.

2. Set Up Jenkins Credentials

- Go to `Manage Jenkins` > `Manage Credentials`.
- Add SSH credentials for the EC2 instance. Use the private key for the user that has access to the EC2 instance.

Step 3: Create a Jenkins Pipeline

1. **Create a New Pipeline Job**:
   - Go to the Jenkins dashboard and click on `New Item`.
   - Name your job (e.g., `WebApp-CI-CD`), select `Pipeline`, and click `OK`.
2. **Define the Pipeline Script**:
   - Under the `Pipeline` section, select `Pipeline script` and start writing the pipeline.

Step 4: Define the Pipeline Stages

Here's an example of a Jenkins pipeline script:

```
pipeline {
    agent any

    environment {
        EC2_USER = 'ec2-user'
        EC2_HOST = 'ec2-instance-public-ip'
        SSH_CREDENTIALS_ID = 'your-ssh-credentials-id'
    }

    stages {
        stage('Clone Repository') {
            steps {
                git branch: 'main', url: 'https://github.com/your-repo/sample-
web-app.git'
            }
        }

        stage('Build') {
            steps {
                echo 'Building the application...'
                // Add your build commands here, e.g., for a Node.js app:
                sh 'npm install'
            }
        }

        stage('Test') {
            steps {
                echo 'Running tests...'
                // Add your test commands here, e.g.:
                sh 'npm test'
            }
        }
```

```
        stage('Deploy') {
            steps {
                sshagent(credentials: [env.SSH_CREDENTIALS_ID]) {
                    echo 'Deploying application...'
                    sh """
                    ssh -o StrictHostKeyChecking=no
${env.EC2_USER}@${env.EC2_HOST} 'mkdir -p /var/www/html'
                    scp -r * ${env.EC2_USER}@${env.EC2_HOST}:/var/www/html/
                    """
                }
            }
        }
    }

    post {
        success {
            echo 'Deployment succeeded!'
        }
        failure {
            echo 'Deployment failed.'
        }
    }
}
```

Explanation of Each Stage:

1. **Clone Repository**:
   - **git branch: 'main', url: 'https://github.com/your-repo/sample-web-app.git'**
     - Pulls the code from the specified Git repository and branch.
2. **Build**:
   - **sh 'npm install'**
     - This step installs the dependencies required for the web application. (Adjust this command based on your application's requirements, e.g., Maven for Java, pip for Python.)
3. **Test**:
   - **sh 'npm test'**
     - Runs the automated tests. If the tests fail, the pipeline stops, and the deployment is not triggered.
4. **Deploy**:
   - **sshagent(credentials: [env.SSH_CREDENTIALS_ID])**
     - Establishes an SSH connection to the EC2 instance using the stored credentials.
   - **ssh -o StrictHostKeyChecking=no ${env.EC2_USER}@${env.EC2_HOST} 'mkdir -p /var/www/html'**
     - Ensures the deployment directory exists on the EC2 instance.
   - **scp -r * ${env.EC2_USER}@${env.EC2_HOST}:/var/www/html/**
     - Copies the application files from Jenkins to the EC2 instance.

Step 5: Run the Pipeline

1. **Trigger the Build**:
   - Go to your Jenkins job and click on `Build Now`.
   - Monitor the console output to see each stage's progress.
2. **Verify the Deployment**:
   - Once the pipeline completes successfully, access the EC2 instance's public IP in a browser to see the deployed application.

Step 6: Automate Triggering (Optional)

- **Webhook Trigger**: You can configure a webhook in your Git repository to trigger the Jenkins pipeline automatically whenever there's a new commit.

Supporting Examples:

- **Automated Tests Example**:
  - If you're using a Node.js application, a basic test could look like this in your `test.js` file:

```
const assert = require('assert');

describe('Sample Test', () => {
  it('should return true', () => {
    assert.equal(true, true);
  });
});
```

  - Running `npm test` will execute this test.

You've now set up a basic CI/CD pipeline using Jenkins to pull code from a Git repository, build the application, run tests, and deploy it to an EC2 instance. By following these steps, you can continuously integrate and deploy your application, ensuring it's always up-to-date and tested.

3. Integrate monitoring and logging

To integrate monitoring and logging for your EC2 instance and set up alerts, you can use Amazon CloudWatch, which is a fully managed service that provides monitoring for AWS resources and applications. Here's how you can achieve this:

Step 1: Set Up Amazon CloudWatch for Monitoring

1.1 Enable Detailed Monitoring on EC2 Instance

- By default, EC2 instances have basic monitoring enabled, which collects data at 5-minute intervals. To get more detailed monitoring (1-minute intervals), you need to enable it:
    1. Go to the **EC2 Dashboard** in the AWS Management Console.
    2. Select your instance.
    3. Click on the **Actions** dropdown, then choose **Monitor and troubleshoot** > **Manage detailed monitoring**.
    4. Enable detailed monitoring.

1.2 Create Custom CloudWatch Alarms

- CloudWatch allows you to set up alarms based on specific thresholds for metrics like CPU usage.

Step 2: Monitor CPU, Memory, and Disk Usage

2.1 Monitoring CPU Usage

- **CPU Utilization** is automatically monitored by CloudWatch. To create an alarm:
    1. Go to the **CloudWatch Dashboard**.
    2. Click on **Alarms** > **Create Alarm**.
    3. Select the **EC2** metric namespace.
    4. Choose the **Per-Instance Metrics** and select your instance's **CPUUtilization** metric.
    5. Set the threshold to 80% (e.g., "Whenever CPU utilization is greater than 80% for 5 minutes").
    6. Click **Next** to configure actions.

2.2 Monitoring Memory and Disk Usage

- By default, CloudWatch does not collect memory and disk metrics for EC2 instances. You need to install and configure the **CloudWatch Agent** to monitor these metrics.

2.2.1 Install CloudWatch Agent on EC2 Instance

1. **SSH into your EC2 instance**:
```
ssh ec2-user@your-ec2-instance-public-ip
```

2. **Install CloudWatch Agent**:
    - On Amazon Linux 2, run:
```
sudo yum install amazon-cloudwatch-agent -y
```
    - On Ubuntu, run:
```
sudo apt-get update
sudo apt-get install amazon-cloudwatch-agent -y
```

3. **Create the CloudWatch Agent Configuration File**:
    - Create a configuration file that includes CPU, memory, and disk monitoring. You can use the CloudWatch Agent Configuration Wizard or manually create a JSON configuration file.

Here's a sample configuration file:

```
{
  "metrics": {
    "metrics_collected": {
      "mem": {
        "measurement": [
          "mem_used_percent"
        ],
        "metrics_collection_interval": 60
      },
      "disk": {
        "measurement": [
          "used_percent"
        ],
        "metrics_collection_interval": 60,
        "resources": [
          "/"
        ]
      },
      "cpu": {
        "measurement": [
          "cpu_usage_idle",
          "cpu_usage_user",
          "cpu_usage_system"
        ],
        "metrics_collection_interval": 60
      }
    }
  }
}
```

4. **Start the CloudWatch Agent**:

```
sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl \
-a fetch-config -m ec2 -c file:/path/to/your-config.json -s
```

2.3 Create Alarms for Memory and Disk Usage

1. Go back to the **CloudWatch Dashboard**.
2. Click on **Alarms** > **Create Alarm**.
3. Choose the **CWAgent** namespace for memory and disk metrics.
4. Follow the steps to set thresholds similar to how you did for CPU utilization.

Step 3: Set Up Email Notifications for Alarms

3.1 Create an SNS Topic

1. Go to the **Simple Notification Service (SNS)** in the AWS Management Console.
2. Click on **Topics** > **Create Topic**.
3. Choose a name for your topic (e.g., `High-CPU-Usage-Alerts`).
4. Create the topic and click on the topic to open it.
5. Click **Create Subscription**.
6. Set the protocol to **Email**, enter your email address, and create the subscription.
7. Confirm the subscription by clicking on the link sent to your email.

3.2 Attach the SNS Topic to CloudWatch Alarms

1. When creating an alarm in CloudWatch, under the **Actions** section, select **Send notification to** and choose your SNS topic.
2. Complete the alarm creation.

Step 4: Verify and Test the Setup

1. **Trigger the Alarm**:
   - You can manually increase CPU usage by running a CPU-intensive process on the EC2 instance to see if the alarm triggers.

```
sudo yum install stress -y
stress --cpu 4 --timeout 600
```

2. **Check Email Alerts**:
   - Once the CPU usage exceeds 80%, CloudWatch will trigger the alarm, and you should receive an email notification.

By following these steps, you've successfully integrated Amazon CloudWatch to monitor CPU, memory, and disk usage on your EC2 instance. You've also set up an alarm to notify you via email if CPU usage exceeds 80%. This setup ensures that you are alerted to potential performance issues and can take corrective actions promptly.
