**Lab Guide: Terraform AWS Deployment with VPC, Subnet, and EC2 Instances**

This lab guide walks you through setting up an AWS Virtual Private Cloud (VPC), a subnet, and three EC2 instances using Terraform. Each EC2 instance will have a unique type and name based on the provided configuration. Follow the steps below to implement the configuration.

**Prerequisites**

* Terraform installed on your local machine.
* An AWS account with valid Access Key and Secret Key.
* Basic understanding of Terraform and AWS services.
* Administrator-level IAM permissions to manage VPC, subnets, and EC2 instances.

**Step 1: Prepare the Lab Environment**

1. **Launch a Terminal:** Open a terminal on your local machine where Terraform is installed.
2. **Create a Directory for the Lab:**
3. mkdir ~/terraform-lab

cd ~/terraform-lab

1. **Create a Terraform Configuration File:** Create a file named main.tf in the directory.
2. **Edit main.tf:** Add the following code to main.tf:

provider "aws" {

region = "us-east-1"

access\_key = "<your-access-key>"

secret\_key = "<your-secret-key>"

}

resource "aws\_vpc" "manual\_vpc" {

cidr\_block = "10.0.0.0/16"

enable\_dns\_support = true

enable\_dns\_hostnames = true

tags = {

Name = "manual-vpc-raman"

}

}

resource "aws\_subnet" "private\_subnet" {

vpc\_id = aws\_vpc.manual\_vpc.id

cidr\_block = "10.0.0.0/24"

availability\_zone = "ap-south-1a"

map\_public\_ip\_on\_launch = false

tags = {

Name = "private-subnet-raman"

}

}

resource "aws\_instance" "private\_instance" {

count = 3

ami = "ami-0187337106779cdf8"

instance\_type = var.type[count.index]

subnet\_id = aws\_subnet.private\_subnet.id

tags = {

Name = "private-${var.names[count.index]}"

}

}

variable "type" {

type = list

default = ["t2.micro", "t2.medium", "t2.large"]

}

variable "names" {

type = list

default = ["dev-server", "stage-server", "prod-server"]

}

**Step 2: Initialize Terraform**

1. **Initialize the Working Directory:** Run the following command in your terminal to initialize Terraform:

terraform init

This command downloads necessary plugins for the AWS provider.

**Step 3: Review the Terraform Plan**

1. **Generate the Execution Plan:** Run the following command to check the resources Terraform will create:

terraform plan

1. **Verify Outputs:** Confirm that the plan matches your desired configuration. The output will list:
   * A VPC named manual-vpc-raman.
   * A private subnet named private-subnet-raman.
   * Three EC2 instances with the names and types specified in the variables:
     + dev-server (t2.micro)
     + stage-server (t2.medium)
     + prod-server (t2.large)

**Step 4: Deploy the Resources**

1. **Apply the Plan:** Run the following command to deploy the resources:

terraform apply

1. **Confirm Changes:** Type yes when prompted to confirm the creation of resources.
2. **Wait for Completion:** Terraform will provision the VPC, subnet, and EC2 instances. Upon success, you will see output confirming resource creation.

**Step 5: Verify Resources in AWS**

1. **Log in to the AWS Console:** Navigate to the AWS Management Console and open the VPC dashboard.
2. **Check the VPC:**
   * Confirm the VPC with CIDR 10.0.0.0/16 and name manual-vpc-raman exists.
3. **Check the Subnet:**
   * In the Subnets section, verify the private subnet 10.0.0.0/24 in the availability zone ap-south-1a.
4. **Check the EC2 Instances:**
   * Navigate to the EC2 dashboard.
   * Ensure there are three instances with the correct types (t2.micro, t2.medium, t2.large) and names (dev-server, stage-server, prod-server).

**Step 6: Clean Up Resources (Optional)**

To avoid incurring costs for the resources created, destroy them when done.

1. **Run Destroy Command:**

terraform destroy

1. **Confirm Destruction:** Type yes to confirm the deletion of resources.
2. **Verify in AWS Console:** Ensure all created resources have been removed from the console.

**Notes:**

* Replace <your-access-key> and <your-secret-key> with your actual AWS credentials.
* Modify variables type and names as needed for different instance configurations.
* Never share or hard-code sensitive information like access keys in production environments. Use secure mechanisms such as AWS credentials file or environment variables.

**Lab Guide: Creating AWS S3 Buckets and EC2 Instances Dynamically with Terraform**

**Objective:**

This lab will guide you through the steps to dynamically create S3 buckets and EC2 instances in AWS using Terraform. You will learn how to use variables, loops, conditionals, and outputs for dynamic resource management.

**Prerequisites:**

1. An AWS account with necessary IAM permissions to create S3 buckets and EC2 instances.
2. Terraform installed on your local machine.
3. Basic understanding of Terraform concepts like variables, resources, and loops.
4. Access to a terminal or IDE for executing Terraform commands.

**Lab Steps:**

**1. Set Up Your Workspace**

**Step 1.1: Create a Directory for Your Project**

mkdir terraform-aws-lab

cd terraform-aws-lab

**Step 1.2: Create a Terraform File**

Create a file named main.tf in the directory.

touch main.tf

**2. Define Variables**

**Step 2.1: Add Variable Blocks to main.tf**

These variables will allow you to configure the project dynamically.

variable "environment" {

description = "Environment for the resources (e.g., dev, test, prod)"

type = string

default = "dev"

}

variable "project\_name" {

description = "Project associated with the resources"

type = string

default = "my-project"

}

variable "s3\_bucket\_names" {

description = "List of S3 buckets to create for the environment"

type = map(string)

default = {

dev = "dev-bucket"

test = "test-bucket"

prod = "prod-bucket"

}

}

variable "instance\_configurations" {

description = "Configurations for instances based on environment"

type = map(map(string))

default = {

dev = { ami = "ami-0c1a7f89451184c8b", instance\_type = "t2.micro" }

test = { ami = "ami-0c1a7f89451184c8b", instance\_type = "t2.small" }

prod = { ami = "ami-0c1a7f89451184c8b", instance\_type = "t2.large" }

}

}

**3. Configure AWS Provider**

**Step 3.1: Add the AWS Provider Configuration**

provider "aws" {

region = "us-east-1"

}

**4. Define S3 Bucket Resources**

**Step 4.1: Add the S3 Bucket Loop**

resource "aws\_s3\_bucket" "buckets" {

for\_each = var.s3\_bucket\_names

bucket = "${each.value}-${var.environment}-${var.project\_name}"

acl = var.environment == "prod" ? "private" : "public-read"

tags = {

Environment = var.environment

Name = each.value

Project = var.project\_name

}

}

This code dynamically creates S3 buckets based on the specified environment and naming conventions.

**5. Define EC2 Instance Resources**

**Step 5.1: Add EC2 Instances Configuration**

resource "aws\_instance" "instances" {

ami = var.instance\_configurations[var.environment]["ami"]

instance\_type = var.instance\_configurations[var.environment]["instance\_type"]

tags = {

Name = "${var.environment}-${var.project\_name}"

Environment = var.environment

Project = var.project\_name

}

}

This block dynamically selects the AMI and instance type based on the environment.

**6. Add Outputs**

**Step 6.1: Add Outputs for Verification**

output "s3\_bucket\_names" {

value = [for i in aws\_s3\_bucket.buckets : i.bucket]

}

output "ec2\_instance\_ids" {

value = [aws\_instance.instances.id]

}

These outputs display the S3 bucket names and EC2 instance IDs after Terraform execution.

**7. Initialize Terraform**

**Step 7.1: Initialize Terraform in Your Directory**

terraform init

This downloads the necessary provider plugins and sets up your workspace.

**8. Plan the Terraform Execution**

**Step 8.1: Run Terraform Plan**

terraform plan

Review the changes that will be made to your AWS environment.

**9. Apply Terraform Configuration**

**Step 9.1: Execute the Terraform Plan**

terraform apply

Type yes when prompted to apply the changes.

**10. Verify Resources**

**Step 10.1: Check the Outputs**

Once the apply command is successful, check the outputs for S3 bucket names and EC2 instance IDs.

terraform output

**Step 10.2: Verify Resources in AWS Console**

* Go to the S3 section in the AWS Management Console to verify the created buckets.
* Navigate to the EC2 dashboard to check the instances.

**Clean-Up**

**Step 11: Destroy Resources**

To avoid unnecessary charges, destroy the resources created during this lab:

terraform destroy

Type yes when prompted.

**Summary**

In this lab, you learned how to:

1. Use variables to define dynamic configurations.
2. Use loops (for\_each) to create multiple resources.
3. Apply conditionals to enforce logic based on environment.
4. Output resource details for verification.

By following this guide, you dynamically created and managed S3 buckets and EC2 instances for different environments with Terraform.

**Lab Guide: Provisioning AWS Resources with Terraform Remote and Local Exec Provisioners**

**Objective**

This lab provides step-by-step instructions to provision AWS resources using Terraform, with a focus on leveraging both remote-exec and local-exec provisioners for configuration and automation.

**Prerequisites**

1. **AWS Account:**
   * IAM user with permissions to manage EC2 instances, security groups, and key pairs.
2. **Key Pair:**
   * Pre-created key pair for SSH. Ensure you download and save the private key securely (e.g., raman.pem).
3. **Terraform Installed:**
   * Install Terraform on your system.
4. **Basic Tools:**
   * Access to a terminal or IDE for executing Terraform commands.
5. **AWS VPC Information:**
   * An existing VPC ID where resources will be deployed.

**Part 1: Terraform with remote-exec Provisioner**

**Overview**

The remote-exec provisioner allows commands to execute directly on the EC2 instance after it is created. Here, we will:

1. Create an EC2 instance with a security group allowing SSH and HTTP access.
2. Install and start the Nginx web server using SSH-based provisioning.

**Steps**

**1. Create Terraform File (remote.tf)**

Create a file named remote.tf and add the following configuration:

provider "aws" {

region = "us-east-1"

access\_key = "<YOUR\_ACCESS\_KEY>" # Replace with your AWS Access Key

secret\_key = "<YOUR\_SECRET\_KEY>" # Replace with your AWS Secret Key

}

resource "aws\_security\_group" "allow\_ssh" {

name = "allow\_ssh"

description = "Allow SSH and HTTP inbound traffic"

vpc\_id = "<YOUR\_VPC\_ID>" # Replace with your VPC ID

ingress {

description = "Allow SSH"

from\_port = 22

to\_port = 22

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

ingress {

description = "Allow HTTP"

from\_port = 80

to\_port = 80

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

egress {

description = "Allow all outbound"

from\_port = 0

to\_port = 65535

protocol = "-1"

cidr\_blocks = ["0.0.0.0/0"]

}

}

resource "aws\_instance" "myec2" {

ami = "ami-0166fe664262f664c" # Amazon Linux AMI

instance\_type = "t2.micro"

key\_name = "remote-key" # Use the name of your key pair

vpc\_security\_group\_ids = [aws\_security\_group.allow\_ssh.id]

tags = {

Name = "raman-web-server"

}

provisioner "remote-exec" {

inline = [

"sudo amazon-linux-extras install -y nginx1.12",

"sudo systemctl start nginx"

]

connection {

type = "ssh"

user = "ec2-user"

private\_key = file("./raman.pem") # Ensure the key file is in this path

host = self.public\_ip

}

}

}

**2. Initialize Terraform**

Run the following command in the directory containing remote.tf:

terraform init

This downloads the necessary provider plugins.

**3. Apply the Configuration**

Execute the configuration:

terraform apply

Type yes when prompted. Terraform will:

1. Create a security group and EC2 instance.
2. Provision the instance using the remote-exec provisioner to install and start Nginx.

**4. Verify Resources**

1. Open the AWS Management Console to confirm the instance is running.
2. Use the public IP of the instance to access the Nginx welcome page in your browser.

**5. Clean-Up**

Destroy the resources after verification to avoid unnecessary charges:

terraform destroy

Type yes to confirm.

**Part 2: Terraform with local-exec Provisioner**

**Overview**

The local-exec provisioner runs commands on your local machine. Here, it will capture the private IP of an EC2 instance and append it to a local file.

**Steps**

**1. Create Terraform File (local\_exec.tf)**

Create a new file named local\_exec.tf and include the following:

provider "aws" {

region = "us-east-2"

}

resource "aws\_instance" "myec2" {

ami = "ami-064ff912f78e3e561" # Ubuntu AMI

instance\_type = "t2.micro"

provisioner "local-exec" {

command = "echo ${aws\_instance.myec2.private\_ip} >> private\_ips.txt"

}

}

**2. Initialize Terraform**

Run:

terraform init

**3. Apply the Configuration**

Run the following command to create the instance and capture its private IP:

terraform apply

Type yes when prompted.

**4. Verify Output**

1. Open the private\_ips.txt file in your working directory to find the private IP of the instance.
2. Verify the instance is running in the AWS Management Console.

**5. Clean-Up**

Destroy the resources after completion:

terraform destroy

Type yes to confirm.

**Summary**

This lab demonstrated:

1. Using remote-exec provisioner to install and configure software on an EC2 instance remotely.
2. Using local-exec provisioner to perform local system tasks based on EC2 instance state.

By integrating these provisioners, you expanded Terraform's capability beyond simple resource management to automation and orchestration.

**Lab Guide: Using Terraform Workspaces to Manage AWS Resources Dynamically**

This detailed lab guide walks through configuring and using Terraform workspaces for provisioning AWS resources dynamically based on different environments.

**Objective**

Demonstrate managing multiple environments (default, dev, prod) using Terraform workspaces, dynamically assigning configurations like instance types and tags.

**Prerequisites**

* **AWS Account**: Ensure you have credentials with sufficient permissions to manage EC2 instances.
* **Terraform Installed**: Verify Terraform is installed.
* **Key Pair (Optional)**: Generate and configure key pairs for SSH, if needed.

**Steps**

**Step 1: Create the Terraform Configuration File**

1. **Create workspaces.tf**: Copy and save the following configuration file in your working directory:

hcl

Copy code

provider "aws" {

region = "us-east-1"

access\_key = "YOUR\_ACCESS\_KEY" # Replace with your AWS access key

secret\_key = "YOUR\_SECRET\_KEY" # Replace with your AWS secret key

}

resource "aws\_instance" "myec2" {

ami = "ami-0187337106779cdf8" # Amazon Linux AMI

instance\_type = lookup(var.instance\_type, terraform.workspace)

tags = {

Name = lookup(var.name, terraform.workspace)

}

}

variable "instance\_type" {

type = map

default = {

default = "t2.nano"

dev = "t2.micro"

prod = "t2.large"

}

}

variable "name" {

type = map

default = {

default = "IT server"

dev = "dev-server"

prod = "prod-server"

}

}

**Step 2: Initialize the Terraform Configuration**

1. **Initialize Terraform**:

bash

Copy code

terraform init

This sets up the Terraform environment and downloads the necessary providers.

1. **View Current Workspace**:

bash

Copy code

terraform workspace show

Confirm default workspace is active initially.

**Step 3: Create and Manage Workspaces**

1. **Create a New Workspace (dev)**:

bash

Copy code

terraform workspace new dev

1. **Create Another Workspace (prod)**:

bash

Copy code

terraform workspace new prod

1. **List Workspaces**:

bash

Copy code

terraform workspace list

This will display:

arduino

Copy code

\* default

dev

prod

1. **Switch Between Workspaces**:

bash

Copy code

terraform workspace select dev

**Step 4: Plan and Apply Configuration**

**For Each Workspace:**

1. **Generate a Terraform Plan**:

bash

Copy code

terraform plan

Observe the ami, instance\_type, and Name tag values. For example:

* + For dev: t2.micro, dev-server.
  + For prod: t2.large, prod-server.

1. **Apply Configuration**:

bash

Copy code

terraform apply

Type yes when prompted to confirm resource creation.

1. **Repeat** for the remaining workspaces (prod and default).

**Step 5: Verify Resources**

1. Log in to the AWS Console.
2. Navigate to **EC2 Instances** and verify the instances are created as per their configurations (instance type and tag).

**Step 6: Cleanup Resources**

1. **Destroy Resources in Each Workspace**:

bash

Copy code

terraform destroy

1. **Switch and Clean Other Workspaces**:

bash

Copy code

terraform workspace select dev

terraform destroy

**Key Notes**

* Dynamic assignment in lookup() allows tailoring resource configurations based on the active workspace.
* Workspaces help manage environment-specific infrastructure (e.g., dev, prod) without changing the core Terraform files.

By following this guide, you gain insights into using workspaces effectively and dynamically managing infrastructure using Terraform.

**Lab Guide: Setting Up an S3 Backend for Terraform State Management**

This lab guide details the steps to configure an S3 bucket for storing Terraform state files securely with versioning enabled and encryption applied. Additionally, you will configure IAM roles to grant necessary permissions.

**Objective**

1. Create an S3 bucket to serve as a backend for Terraform state files.
2. Enable versioning for improved availability and state file rollback capability.
3. Secure the state file with server-side encryption.
4. Modify IAM roles on an EC2 instance to include permissions for Terraform.
5. Configure Terraform to use S3 as the backend.

**Prerequisites**

* **AWS Account:** Ensure you have administrative access.
* **Terraform Installed:** Ensure Terraform is installed on your local system.
* **IAM Role:** Admin-level IAM role for managing S3 and Terraform state.
* **EC2 Instance:** An EC2 instance with an existing IAM role that will interact with the S3 bucket.
* **CLI Tools:** AWS CLI for verifying bucket and IAM role configurations.

**Steps**

**Step 1: Create an S3 Bucket for Terraform State File**

1. **Login to the AWS Console**: Navigate to the S3 service.
2. **Create a New Bucket**:
   * Click **Create Bucket**.
   * Configure the following:
     + **Bucket Name:** cisco-statefile
     + **Region:** us-east-1
3. **Enable Bucket Versioning**:
   * After creating the bucket, select it from the S3 dashboard.
   * Go to the **Properties** tab and enable **Bucket Versioning**.
4. **Enable Server-Side Encryption**:
   * In the bucket’s **Properties** tab, enable **Default Encryption**.
   * Use **AES-256 (SSE-S3)** or a KMS-managed key for encryption.

**Step 2: Assign IAM Role Permissions**

1. **Modify IAM Role for EC2 Instance**:
   * Go to **EC2 Dashboard** > **Instances**.
   * Select the instance where Terraform will run.
   * Click **Actions** > **Security** > **Modify IAM Role**.
   * Attach an IAM role with the following permissions:
     + s3:PutObject
     + s3:GetObject
     + s3:ListBucket
     + s3:DeleteObject
2. **Example IAM Policy for Terraform S3 Backend**:
3. {
4. "Version": "2012-10-17",
5. "Statement": [
6. {
7. "Effect": "Allow",
8. "Action": [
9. "s3:ListBucket"
10. ],
11. "Resource": "arn:aws:s3:::cisco-statefile"
12. },
13. {
14. "Effect": "Allow",
15. "Action": [
16. "s3:PutObject",
17. "s3:GetObject",
18. "s3:DeleteObject"
19. ],
20. "Resource": "arn:aws:s3:::cisco-statefile/terraform/\*"
21. }
22. ]

}

**Step 3: Configure Terraform Backend**

1. **Create a backend.tf File**: Add the following Terraform configuration to your project:
2. terraform {
3. backend "s3" {
4. bucket = "cisco-statefile"
5. key = "terraform/state"
6. region = "us-east-1"
7. }

}

1. **Initialize Terraform**: Run the following command to initialize the S3 backend:

terraform init

Terraform will configure itself to use the S3 bucket as the state file location.

1. **Verify Initialization**:
   * Check the S3 bucket to confirm the terraform/state file has been created.
   * Ensure encryption and versioning are applied by reviewing the S3 bucket settings.

**Step 4: Additional Terraform Commands**

**Apply a Sample Configuration**

1. Create a main.tf file with a basic resource:
2. provider "aws" {
3. region = "us-east-1"
4. }
5. resource "aws\_s3\_bucket" "example" {
6. bucket = "example-terraform-bucket"
7. acl = "private"

}

1. Plan and Apply:
2. terraform plan

terraform apply

1. Check the terraform/state file in the cisco-statefile S3 bucket to verify the state file is being updated correctly.

**Clean-Up Resources**

After verifying the setup, destroy the resources to avoid unnecessary costs:

terraform destroy

**Verification and Troubleshooting**

* **Bucket Versioning:** Verify in the AWS Console > S3 > Bucket Properties > Versioning.
* **Encryption:** Confirm encryption settings under Bucket Properties > Default Encryption.
* **State File Updates:** Check that the terraform/state file is being updated with changes.
* **IAM Role Issues:** If Terraform cannot access the S3 bucket, verify the IAM role's permissions.

**Summary**

This lab demonstrates how to:

1. Configure an S3 bucket for storing Terraform state files with versioning and encryption.
2. Modify IAM roles to ensure secure access.
3. Initialize and use the S3 backend in Terraform projects for state management.

With this configuration, Terraform state is secured, resilient, and enables multi-user collaboration in a robust manner.