Terraform Projects Part-1

1. Cloud Networking & VPC

Project 1. VPC Peering setting up VPC Peering services across different cloud platforms using Terraform.

Project 2. VPC Peering and Transit Gateway Setup

Project 3. Custom VPC Networking

Project 4. VPC Network Design

Project 5 Private Link and Service Endpoints

Project 6. Automated Networking Setup with Terraform

2. High Availability & Disaster Recovery

Project 1. High Availability Architecture

Project 2. Disaster Recovery Setup

Project 3 Disaster Recovery Simulation

Project 4. Disaster Recovery and High Availability Setup

Project 5. Disaster Recovery Setup

Project 6 Automated Disaster Recovery for Cloud Infrastructure

---

3. Serverless

Project 1. Serverless Application Deployment

Project 2. Serverless Infrastructure with

Project 3. Serverless Architecture with AWS API Gateway & Lambda

Project 4. Serverless Event-Driven Architecture

Project 5. Event-Driven Serverless Microservices

---

4. Hybrid Cloud

Project 1. Hybrid Cloud Infrastructure

Project 2. Hybrid Cloud Setup

Project 3. Infrastructure as Code for Docker Swarm

Project 4. Cross-Cloud Application Deployment

Project 5. Cross-Account Resource Sharing

---

5. Kubernetes & Containers

Project 1 Kubernetes Cluster with Terraform

Project 2. Managed Kubernetes Infrastructure

Project 3. Managed Kubernetes with Helm

Project 4. Terraform for Kubernetes Multi-Cluster Management

Project 5 Terraform for Kubernetes Multi-Cluster Management

Project 6 Terraform for Kubernetes with Istio Service Mesh

---

6. CI/CD Pipeline

Project 1. CI/CD Pipeline for Infrastructure

Project 2. Terraform for CI/CD Tools

Project 3. GitOps with Terraform and ArgoCD

Project 4 Terraform for Serverless CI/CD Pipeline

Project 5. Terraform with Ansible Integration

Terraform Projects Part-1

1. Cloud Networking & VPC

Project 1. VPC Peering setting up VPC Peering services across different cloud platforms using Terraform.

Introduction to VPC Peering

- **Definition** VPC Peering allows two Virtual Private Clouds (VPCs) or Virtual Networks (VNets) to communicate with each other using private IP addresses.
- **Use Case** Seamlessly connect applications and resources across isolated networks for secure communication without going over the public internet.

Set Up Terraform Environment

Install Terraform Download and install Terraform for your operating system from the official website.

set up the credentials for Terraform to interact with different cloud providers, you need to authenticate Terraform with the cloud service. Here's a

Configure credentials for AWS:

1. AWS (Amazon Web Services)

For AWS, you have two main ways to configure your credentials:

- **~/.aws/credentials File**: This is the most common method where you store your AWS access key and secret key.

  Ensure that you have the AWS CLI installed.

  Run `aws configure` to set your access key, secret key, region, and output format.

  Your credentials will be stored in the `~/.aws/credentials` file.

  Terraform will automatically use the credentials from this file.

  Example:

```
aws configure

AWS Access Key ID [None]: YOUR_ACCESS_KEY_ID

AWS Secret Access Key [None]: YOUR_SECRET_ACCESS_KEY

Default region name [None]: us-west-2

Default output format [None]: json
```

  Environment Variables: You can also set the AWS access key and secret key using environment variables in your terminal session.

```
export AWS_ACCESS_KEY_ID=YOUR_ACCESS_KEY_ID

export AWS_SECRET_ACCESS_KEY=YOUR_SECRET_ACCESS_KEY

export AWS_DEFAULT_REGION=us-west-2
```

Configuration

1. **Create Two VPCs**
    - VPC A CIDR block 10.0.0.0/16
    - VPC B CIDR block 10.1.0.0/16

2. **Establish VPC Peering** between VPC A and VPC B.

Terraform Code

```
provider "aws" {

  region = "us-west-2"

}

resource "aws_vpc" "vpc_a" {

  cidr_block = "10.0.0.0/16"

}


resource "aws_vpc" "vpc_b" {

  cidr_block = "10.1.0.0/16"

}

resource "aws_vpc_peering_connection" "peer_connection" {

  vpc_id      = aws_vpc.vpc_a.id

  peer_vpc_id = aws_vpc.vpc_b.id

  auto_accept = true

  tags = {

    Name = "aws-vpc-peer"

  }
```

}

Steps to Apply

- Save the code in main.tf.
- Run terraform init to initialize the working directory.
- Run terraform apply to deploy the resources.

Azure VNet Peering

Configure credentials for Azure:

For Azure, you can authenticate using either the Azure CLI or by creating a service principal (a special type of Azure AD application).

- **az login** (for Azure CLI):

  If you already have Azure CLI installed, you can run `az login` to authenticate with your Azure account.

  This will open a browser window for you to log in to your Azure account. After logging in, the credentials will be available for Terraform to use.
  Example:
  ```
  az login
  ```

- **Service Principal Authentication**:

  A service principal is an identity that you can create to authenticate applications with Azure resources.

  You can create a service principal using the `az ad sp create-for-rbac` command.

  Store the output from this command (client ID, tenant ID, and client secret) in your environment variables.
  Example:

  ```
  export ARM_CLIENT_ID="your-client-id"

  export ARM_CLIENT_SECRET="your-client-secret"
  ```

```
export ARM_SUBSCRIPTION_ID="your-subscription-id"

export ARM_TENANT_ID="your-tenant-id"
```

Configuration

1. **Create Two VNets**
    - VNet A Address space 10.0.0.0/16
    - VNet B Address space 10.1.0.0/16
2. **Establish VNet Peering** between VNet A and VNet B.

Terraform Code

```
provider "azurerm" {

  features {}

}


resource "azurerm_resource_group" "rg" {

  name     = "rg-vnet-peering"

  location = "East US"

}


resource "azurerm_virtual_network" "vnet_a" {

  name                = "vnet-a"

  location            = azurerm_resource_group.rg.location

  address_space       = ["10.0.0.0/16"]

  resource_group_name = azurerm_resource_group.rg.name
```

```
}

resource "azurerm_virtual_network" "vnet_b" {
  name                = "vnet-b"
  location            = azurerm_resource_group.rg.location
  address_space       = ["10.1.0.0/16"]
  resource_group_name = azurerm_resource_group.rg.name
}

resource "azurerm_virtual_network_peering" "peering_a_to_b" {
  name                      = "vnet-a-to-vnet-b"
  resource_group_name       = azurerm_resource_group.rg.name
  virtual_network_name      = azurerm_virtual_network.vnet_a.name
  remote_virtual_network_id = azurerm_virtual_network.vnet_b.id
  allow_virtual_network_access = true
  allow_forwarded_traffic      = true
}

resource "azurerm_virtual_network_peering" "peering_b_to_a" {
  name                      = "vnet-b-to-vnet-a"
  resource_group_name       = azurerm_resource_group.rg.name
  virtual_network_name      = azurerm_virtual_network.vnet_b.name
  remote_virtual_network_id = azurerm_virtual_network.vnet_a.id
```

```
  allow_virtual_network_access = true

  allow_forwarded_traffic      = true

}
```

Steps to Apply

- Save the code in main.tf.
- Run terraform init.
- Run terraform apply.

GCP VPC Peering

Credential Configuration of GCP:

For GCP, you need to use a service account key in JSON format.

- **Service Account Key**:

  Go to the **Google Cloud Console**.

  Create a **service account** and download the JSON key file.

  Set the environment variable `GOOGLE_APPLICATION_CREDENTIALS` to point to this JSON key file.
  Example:

  ```
  export GOOGLE_APPLICATION_CREDENTIALS="/path/to/your-service-account-key.json"
  ```

Terraform Configuration:

1. **Create Two VPCs**
   - VPC A Disable auto subnet creation.
   - VPC B Disable auto subnet creation.

2. **Establish VPC Peering** between VPC A and VPC B.

Terraform Code

```
provider "google" {
  region = "us-central1"
}
resource "google_compute_network" "vpc_a" {
  name                    = "vpc-a"
  auto_create_subnetworks = false
}
resource "google_compute_network" "vpc_b" {
  name                    = "vpc-b"
  auto_create_subnetworks = false
}
resource "google_compute_network_peering" "peer_a_to_b" {
  name         = "peer-a-to-b"
  network      = google_compute_network.vpc_a.name
  peer_network  = google_compute_network.vpc_b.name
  auto_create_routes = true
}
resource "google_compute_network_peering" "peer_b_to_a" {
  name         = "peer-b-to-a"
  network      = google_compute_network.vpc_b.name
  peer_network  = google_compute_network.vpc_a.name
```

```
  auto_create_routes = true

}
```

Steps to Apply

- Save the code in main.tf.
- Run terraform init.
- Run terraform apply.

Oracle Cloud VCN Peering

This will allow Terraform to authenticate using the service account.

For Oracle Cloud, you authenticate using an API key and tenancy details.

- **API Key Authentication**:

  Log in to the Oracle Cloud Console and navigate to **Identity & Security** > **Users**.

  Select your user and then create a new API key under **API Keys**.

  Download the private key (the `.pem` file).

  Set up the environment variables for your OCI credentials:

  - `OCI_CLI_AUTH` (authentication type)
  - `OCI_CONFIG_FILE` (path to your config file)
  - `OCI_PRIVATE_KEY_PATH` (path to the private key `.pem` file)
  - `OCI_TENANCY_OCID`, `OCI_USER_OCID`, `OCI_REGION`, etc.

  Example:

```
export OCI_CLI_AUTH=api_key

export OCI_PRIVATE_KEY_PATH="/path/to/private-key.pem"

export OCI_TENANCY_OCID="your-tenancy-id"

export OCI_USER_OCID="your-user-id"
```

```
export OCI_REGION="us-phoenix-1"
```

Terraform Configuration:

1. **Create Two VCNs**
    - VCN A CIDR block 10.0.0.0/16
    - VCN B CIDR block 10.1.0.0/16
2. **Establish VCN Peering** between VCN A and VCN B.

Terraform Code

```
provider "oci" {
  region = "us-phoenix-1"
}
resource "oci_identity_compartment" "compartment" {
  name          = "compartment-vcn"
  description   = "Compartment for VCN Peering"
  compartment_id = var.tenancy_ocid
}

resource "oci_core_virtual_network" "vcn_a" {
  compartment_id = oci_identity_compartment.compartment.id
  cidr_block     = "10.0.0.0/16"
  display_name   = "vcn-a"
}
resource "oci_core_virtual_network" "vcn_b" {
```

```
  compartment_id = oci_identity_compartment.compartment.id
  cidr_block     = "10.1.0.0/16"
  display_name   = "vcn-b"
}
resource "oci_core_virtual_network_peering" "vcn_peer" {
  compartment_id = oci_identity_compartment.compartment.id
  vcn_id         = oci_core_virtual_network.vcn_a.id
  peer_vcn_id    = oci_core_virtual_network.vcn_b.id
  display_name   = "vcn-peer"
  peer_region_name = "us-phoenix-1"
  allow_remote_vpc_ipv6 = true
}
```

Steps to Apply

- Save the code in main.tf.
- Run terraform init.
- Run terraform apply.

Conclusion

- **Outcome** all four platforms are now connected securely using private IP addresses.
- **Next Steps** Verify peering configurations test connectivity and monitor traffic as required.

Project 2. VPC Peering and Transit Gateway Setup

Automate VPC peering connections and use AWS Transit Gateway to manage network traffic between multiple VPCs across different regions.

Here is a general approach for Terraform configurations for VPC peering and network traffic management across AWS Azure Google Cloud and Oracle Cloud

1. AWS VPC Peering and Transit Gateway

VPC Peering Setup (Basic)

1. **Initialize AWS Provider**
    - Define the AWS provider with the desired region.

```
provider "aws" {
  region = "us-east-1"
}
```

Create Two VPCs

- VPC 1 CIDR 10.0.0.0/16
- VPC 2 CIDR 10.1.0.0/16

```
resource "aws_vpc" "vpc_1" {
  cidr_block = "10.0.0.0/16"
  enable_dns_support = true
  enable_dns_hostnames = true
}
resource "aws_vpc" "vpc_2" {
  cidr_block = "10.1.0.0/16"
  enable_dns_support = true
  enable_dns_hostnames = true
}
```

Establish VPC Peering

- Create a peering connection between the two VPCs.

```
resource "aws_vpc_peering_connection" "peer" {
  vpc_id        = aws_vpc.vpc_1.id
  peer_vpc_id   = aws_vpc.vpc_2.id
  auto_accept   = true
  peer_region   = "us-west-1"
}
```

Update Route Tables

- Add routes to allow traffic between VPCs.

```
resource "aws_route" "vpc_1_to_vpc_2" {
  route_table_id         = aws_vpc.vpc_1.main_route_table_id
  destination_cidr_block = "10.1.0.0/16"
  vpc_peering_connection_id = aws_vpc_peering_connection.peer.id
}
resource "aws_route" "vpc_2_to_vpc_1" {
  route_table_id         = aws_vpc.vpc_2.main_route_table_id
  destination_cidr_block = "10.0.0.0/16"
  vpc_peering_connection_id = aws_vpc_peering_connection.peer.id
}
```

Transit Gateway Setup

1. **Create a Transit Gateway**
    - Transit Gateway acts as a central hub for multiple VPCs.

```
resource "aws_ec2_transit_gateway" "example" {
  description = "My Transit Gateway"
}
```

Attach VPCs to Transit Gateway

- Connect each VPC to the Transit Gateway.

```
resource "aws_ec2_transit_gateway_vpc_attachment" "attachment_1" {
  subnet_ids         = [aws_subnet.subnet_1.id]
  transit_gateway_id = aws_ec2_transit_gateway.example.id
  vpc_id             = aws_vpc.vpc_1.id
}
resource "aws_ec2_transit_gateway_vpc_attachment" "attachment_2" {
  subnet_ids         = [aws_subnet.subnet_2.id]
  transit_gateway_id = aws_ec2_transit_gateway.example.id
  vpc_id             = aws_vpc.vpc_2.id
}
```

Update Route Tables for Transit Gateway

- Define routing rules for Transit Gateway traffic.

```
resource "aws_route" "route_to_vpc_1" {
  route_table_id         = aws_vpc.vpc_2.main_route_table_id
```

```
  destination_cidr_block = "10.0.0.0/16"
  transit_gateway_id     = aws_ec2_transit_gateway.example.id
}
```

---

2. Azure VNet Peering

1. **Initialize Azure Provider**
    - Define the Azure provider.

```
provider "azurerm" {
  features {}
}
```

Create Virtual Networks

- Define two VNets with different address spaces.

```
resource "azurerm_virtual_network" "vnet_1" {
  name                = "vnet-1"
  location            = "East US"
  resource_group_name = azurerm_resource_group.rg.name
  address_space       = ["10.0.0.0/16"]
}

resource "azurerm_virtual_network" "vnet_2" {
  name                = "vnet-2"
```

```
  location            = "East US"
  resource_group_name = azurerm_resource_group.rg.name
  address_space       = ["10.1.0.0/16"]
}
```

Create VNet Peering

- Allow traffic between VNets.

```
resource "azurerm_virtual_network_peering" "peer" {
  name                         = "vnet1-to-vnet2"
  resource_group_name          = azurerm_resource_group.rg.name
  virtual_network_name         = azurerm_virtual_network.vnet_1.name
  remote_virtual_network_id    = azurerm_virtual_network.vnet_2.id
  allow_virtual_network_access  = true
  allow_forwarded_traffic      = true
}
```

Define Routes

- Add routing rules for inter-VNet communication.

```
resource "azurerm_route" "route_to_vnet_2" {
  name                        = "route-to-vnet2"
  resource_group_name           = azurerm_resource_group.rg.name
  route_table_name              = azurerm_route_table.rt.name
  address_prefix                = "10.1.0.0/16"
```

```
  next_hop_type                = "VirtualNetworkPeering"
}
```

---

3. Google Cloud VPC Peering

Initialize GCP Provider

- Specify the GCP project and region.

```
provider "google" {
  project = "your-gcp-project"
  region  = "us-central1"
}
```

Create Two VPCs

- Define VPCs with no auto subnetwork creation.

```
resource "google_compute_network" "vpc_1" {
  name                    = "vpc-1"
  auto_create_subnetworks = false
}
resource "google_compute_network" "vpc_2" {
  name                    = "vpc-2"
  auto_create_subnetworks = false
}
```

Establish VPC Peering

- Set up peering between the two VPCs.

```
resource "google_compute_network_peering" "peer" {
  name         = "vpc-1-to-vpc-2"
  network      = google_compute_network.vpc_1.name
  peer_network = google_compute_network.vpc_2.self_link
  auto_create_routes = true
}
```

---

4. Oracle Cloud VCN Peering

Initialize Oracle Provider

- Specify the Oracle Cloud region.

```
provider "oci" {
  region = "us-phoenix-1"
}
```

Create Two VCNs

- Define separate Virtual Cloud Networks.

```
resource "oci_core_virtual_network" "vcn_1" {
  compartment_id = oci_identity_compartment.compartment.id
  display_name   = "vcn-1"
```

```
  cidr_block     = "10.0.0.0/16"
}

resource "oci_core_virtual_network" "vcn_2" {
  compartment_id = oci_identity_compartment.compartment.id
  display_name   = "vcn-2"
  cidr_block     = "10.1.0.0/16"
}
```

Establish VCN Peering

- Connect the two VCNs.

```
resource "oci_core_vcn_peering" "peer" {
  compartment_id = oci_identity_compartment.compartment.id
  vcn_id         = oci_core_virtual_network.vcn_1.id
  peer_vcn_id    = oci_core_virtual_network.vcn_2.id
}
```

Project 3. Custom VPC Networking

Custom VPC (Virtual Private Cloud) networking allows you to control and configure your cloud network environment within public cloud providers like AWS GCP or Azure. It provides a logically isolated network for deploying resources securely.

Create custom VPCs with subnets route tables NAT and Internet gateways.

Step-by-Step Guide to Implement Custom VPC Networking

Prerequisites

- Installed and configured Terraform.
- Cloud provider credentials set up (AWS Azure GCP Oracle).
- Basic understanding of VPC networking components (subnets gateways etc.).

---

1. AWS Custom VPC Setup

Step 1 Define the provider in Terraform

```
provider "aws" {
  region = "us-east-1"
}
```

Step 2 Create a VPC

```
resource "aws_vpc" "custom_vpc" {
  cidr_block = "10.0.0.0/16"
  enable_dns_support = true
  enable_dns_hostnames = true
  tags = { Name = "CustomVPC" }
}
```

Step 3 Create subnets

```
resource "aws_subnet" "public_subnet" {
  vpc_id            = aws_vpc.custom_vpc.id
  cidr_block        = "10.0.1.0/24"
  map_public_ip_on_launch = true
```

```
  availability_zone = "us-east-1a"
}
```

Step 4 Add an Internet Gateway and attach it to the VPC

```
resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.custom_vpc.id
}
```

Step 5 Create route tables and associate them with subnets

```
resource "aws_route_table" "public_routes" {
  vpc_id = aws_vpc.custom_vpc.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
  }
}
resource "aws_route_table_association" "public_subnet_assoc" {
  subnet_id      = aws_subnet.public_subnet.id
  route_table_id = aws_route_table.public_routes.id
}
```

---

2. Azure Custom VPC Setup

Step 1 Define the provider

```
provider "azurerm" {

  features {}

}
```

Step 2 Create a Virtual Network (VNet)

```
resource "azurerm_virtual_network" "custom_vnet" {

  name                = "CustomVNet"

  location            = "East US"

  resource_group_name = azurerm_resource_group.main.name

  address_space       = ["10.0.0.0/16"]

}
```

Step 3 Create subnets

```
resource "azurerm_subnet" "public_subnet" {

  name                 = "PublicSubnet"

  resource_group_name  = azurerm_resource_group.main.name

  virtual_network_name = azurerm_virtual_network.custom_vnet.name

  address_prefixes     = ["10.0.1.0/24"]

}
```

---

3. GCP Custom VPC Setup

Step 1 Define the provider

```
provider "google" {
```

```
  project = "your-gcp-project-id"
  region  = "us-central1"
}
```

Step 2 Create a VPC

```
resource "google_compute_network" "custom_vpc" {
  name = "custom-vpc"
  auto_create_subnetworks = false
}
```

Step 3 Add subnets

```
resource "google_compute_subnetwork" "public_subnet" {
  name          = "public-subnet"
  ip_cidr_range = "10.0.1.0/24"
  network       = google_compute_network.custom_vpc.id
  region        = "us-central1"
}
```

---

4. Oracle Cloud Custom VPC Setup

Step 1 Define the provider

```
provider "oci" {
  region = "us-ashburn-1"
}
```

Step 2 Create a Virtual Cloud Network (VCN)

```
resource "oci_core_vcn" "custom_vcn" {

  cidr_block = "10.0.0.0/16"

  display_name = "CustomVCN"

}
```

Step 3 Add subnets

```
resource "oci_core_subnet" "public_subnet" {

  vcn_id          = oci_core_vcn.custom_vcn.id

  cidr_block      = "10.0.1.0/24"

  display_name    = "PublicSubnet"

  availability_domain = "UocmUS-ASHBURN-AD-1"

}
```

Summary

Each cloud provider has its own syntax and configuration for creating custom VPCs. Terraform allows you to manage these configurations using a consistent workflow. This project involves creating

- VPC/VNet or equivalent.
- Subnets for network segmentation.
- Gateways (Internet or NAT) for external communication.
- Routing rules to control traffic flow.

Using Terraform simplifies the deployment of infrastructure across multiple providers ensuring reproducibility and scalability.

Project 4. VPC Network Design

Use Terraform to design and manage VPCs subnets routing tables and VPN connections for secure network infrastructure.

Introduction to VPC Network Design with Terraform

A Virtual Private Cloud (VPC) is a fundamental building block in cloud networking providing a secure and isolated environment for deploying resources. Terraform an Infrastructure as Code (IaC) tool simplifies the creation configuration and management of VPCs across various cloud providers. This project focuses on using Terraform to design and manage VPCs subnets route tables and VPN connections in AWS Azure Google Cloud Platform (GCP) and Oracle Cloud Infrastructure (OCI). By following this guide you will learn how to establish secure network infrastructure efficiently and consistently across multiple cloud platforms.

Step-by-Step Instructions for Each Cloud Provider

AWS VPC Network Design

1. **Pre-requisites**
    - Install Terraform.
    - Configure AWS CLI and set up credentials.
    - Have an IAM user with sufficient permissions for VPC management.
2. **Create the Terraform Configuration**
    - Initialize the Terraform configuration file (main.tf) with the AWS provider.
    - Define resources for VPC subnets route tables and VPN connections.

Terraform Code Example

```
provider "aws" {

  region = "us-west-1"

}



resource "aws_vpc" "main" {

  cidr_block = "10.0.0.0/16"

}
```

```
resource "aws_subnet" "subnet" {
  vpc_id            = aws_vpc.main.id
  cidr_block        = "10.0.1.0/24"
  availability_zone = "us-west-1a"
}

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

resource "aws_route_table" "route_table" {
  vpc_id = aws_vpc.main.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
  }
}
```

3. **Run Terraform Commands**
    - terraform init Initialize the configuration.
    - terraform plan Preview the resources.
    - terraform apply Deploy the infrastructure.

Azure VPC Network Design

1. **Pre-requisites**
    - Install Terraform.
    - Configure Azure CLI and set up credentials.
    - Create a service principal with required permissions.
2. **Create the Terraform Configuration**
    - Initialize the Terraform configuration file (main.tf) with the Azure provider.
    - Define resources for VNet subnets and VPN gateways.

Terraform Code Example

```
provider "azurerm" {

  features {}

}



resource "azurerm_resource_group" "rg" {

  name     = "example-resources"

  location = "East US"

}



resource "azurerm_virtual_network" "vnet" {

  name                = "example-vnet"

  address_space       = ["10.0.0.0/16"]

  location            = azurerm_resource_group.rg.location

  resource_group_name = azurerm_resource_group.rg.name

}
```

```
resource "azurerm_subnet" "subnet" {
  name                 = "example-subnet"
  resource_group_name  = azurerm_resource_group.rg.name
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefixes     = ["10.0.1.0/24"]
}
```

3. **Run Terraform Commands**
    - terraform init
    - terraform plan
    - terraform apply

GCP VPC Network Design

1. **Pre-requisites**
    - Install Terraform.
    - Configure Google Cloud SDK and authenticate.
    - Enable required APIs in GCP.
2. **Create the Terraform Configuration**
    - Initialize the Terraform configuration file (main.tf) with the GCP provider.
    - Define resources for VPC subnets and routes.

Terraform Code Example

```
provider "google" {
  project = "your-project-id"
  region  = "us-central1"
}

resource "google_compute_network" "vpc_network" {
```

```
  name                    = "example-vpc"
  auto_create_subnetworks = false
}

resource "google_compute_subnetwork" "subnet" {
  name          = "example-subnet"
  ip_cidr_range = "10.0.1.0/24"
  region        = "us-central1"
  network       = google_compute_network.vpc_network.id
}
```

3. **Run Terraform Commands**
    - terraform init
    - terraform plan
    - terraform apply

Oracle Cloud Infrastructure (OCI) VPC Network Design

1. **Pre-requisites**
    - Install Terraform.
    - Configure OCI CLI and obtain authentication keys.
    - Set up an OCI tenancy with proper permissions.
2. **Create the Terraform Configuration**
    - Initialize the Terraform configuration file (main.tf) with the OCI provider.
    - Define resources for VCN subnets and route tables.

Terraform Code Example

```
provider "oci" {
```

```
  tenancy_ocid = "ocid1.tenancy.oc1..xxxx"
  user_ocid    = "ocid1.user.oc1..xxxx"
  fingerprint  = "your-fingerprint"
  private_key_path = "~/.oci/oci_api_key.pem"
  region       = "us-ashburn-1"
}

resource "oci_core_vcn" "vcn" {
  cidr_block     = "10.0.0.0/16"
  display_name   = "example-vcn"
  compartment_id = "ocid1.compartment.oc1..xxxx"
}

resource "oci_core_subnet" "subnet" {
  cidr_block     = "10.0.1.0/24"
  display_name   = "example-subnet"
  vcn_id         = oci_core_vcn.vcn.id
  compartment_id = "ocid1.compartment.oc1..xxxx"
}
```

3. **Run Terraform Commands**
    - terraform init
    - terraform plan
    - terraform apply

Conclusion

By following these steps you can design and manage VPCs subnets and routing configurations across AWS Azure GCP and OCI using Terraform. This project not only provides a consistent approach to managing cloud resources but also ensures scalability and security in network infrastructure.

Project 5 Private Link and Service Endpoints

This project demonstrates how to enable **secure connectivity** using Private Link or Service Endpoints across multiple cloud providers

- **AWS PrivateLink** Secure access to AWS services or your applications over private connectivity.
- **Azure Private Endpoint** Securely connect to Azure services or resources in a virtual network.
- **GCP Private Service Connect** Provide private connectivity to Google services or your services hosted in GCP.
- **Oracle Private Endpoint** Establish secure access to Oracle services or custom endpoints.

Below are step-by-step Terraform configurations for each provider and the required IAM permissions.

AWS PrivateLink

Terraform Configuration

```
provider "aws" {

  region = "us-east-1"

}


resource "aws_vpc_endpoint" "private_link" {
```

```
  vpc_id             = "vpc-123456"
  service_name       = "com.amazonaws.us-east-1.s3"
  vpc_endpoint_type  = "Interface"
  subnet_ids         = ["subnet-123456" "subnet-789012"]

  private_dns_enabled = true
}
```

IAM Permissions

json

```
{
  "Version" "2012-10-17"
  "Statement" [
    {
      "Effect" "Allow"
      "Action" [
        "ec2CreateVpcEndpoint"
        "ec2DescribeVpcEndpoints"
        "ec2ModifyVpcEndpoint"
        "ec2DeleteVpcEndpoints"
        "ec2DescribeSubnets"
        "ec2DescribeVpcs"
```

```
      "ec2DescribeSecurityGroups"
      "ec2ModifySecurityGroups"
      "ec2DescribeRouteTables"
    ]
    "Resource" "*"
   }
 ]
}
```

Azure Private Endpoint

Terraform Configuration

```
provider "azurerm" {
  features = {}
}

resource "azurerm_private_endpoint" "example" {
  name                = "example-private-endpoint"
  location            = "East US"
  resource_group_name = "example-rg"
  subnet_id           =
"/subscriptions/<subscription-id>/resourceGroups/example-rg/providers/Microsoft.Network/virtualNetworks/example-vnet/subnets/private-subnet"
```

```
  private_service_connection {
    name                           = "connection-to-blob"
    private_connection_resource_id =
"/subscriptions/<subscription-id>/resourceGroups/example-rg/providers/Microsoft.Storag
e/storageAccounts/exampleaccount"
    is_manual_connection           = false
    subresource_names              = ["blob"]
  }
}
```

IAM Permissions

- **Contributor** role on the resource group containing the private endpoint.
- **Custom role (optional)**

```
json
{
  "Name" "Private Endpoint Contributor"
  "Actions" [
    "Microsoft.Network/privateEndpoints/*"
    "Microsoft.Network/virtualNetworks/subnets/*"
    "Microsoft.Storage/storageAccounts/*"
    "Microsoft.Resources/subscriptions/resourceGroups/read"
  ]
  "AssignableScopes" [
    "/subscriptions/{subscription-id}/resourceGroups/{resource-group}"
  ]
}
```

GCP Private Service Connect

Terraform Configuration

```
provider "google" {
  project = "my-gcp-project"
  region  = "us-central1"
}

resource "google_compute_network" "network" {
  name = "example-network"
}

resource "google_compute_global_address" "psc_address" {
  name         = "psc-address"
  purpose      = "PRIVATE_SERVICE_CONNECT"
  address_type  = "INTERNAL"
  network      = google_compute_network.network.self_link
}
resource "google_compute_forwarding_rule" "psc_rule" {
  name                  = "psc-rule"
  load_balancing_scheme = "INTERNAL"
  backend_service      = "<backend-service-url>"
```

```
  ip_address           = google_compute_global_address.psc_address.address
  network              = google_compute_network.network.self_link
}
```

IAM Permissions

- **roles/compute.networkAdmin** Manage networks and subnets.
- **roles/compute.globalAddressAdmin** Manage global addresses.
- **roles/compute.forwardingRuleAdmin** Manage forwarding rules.

```
json
{
  "bindings" [
    {
      "role" "roles/compute.networkAdmin"
      "members" ["useryour-email@example.com"]
    }
    {
      "role" "roles/compute.globalAddressAdmin"
      "members" ["useryour-email@example.com"]
    }
    {
      "role" "roles/compute.forwardingRuleAdmin"
      "members" ["useryour-email@example.com"]
    }
  ]
}
```

Oracle Private Endpoint

Terraform Configuration

```
provider "oci" {
  region = "us-ashburn-1"
}

resource "oci_core_vcn" "example_vcn" {
  cidr_block     = "10.0.0.0/16"
  compartment_id = "<compartment-id>"
  display_name   = "example-vcn"
}

resource "oci_core_subnet" "example_subnet" {
  compartment_id = oci_core_vcn.example_vcn.compartment_id
  vcn_id         = oci_core_vcn.example_vcn.id
  cidr_block     = "10.0.1.0/24"
  display_name   = "example-subnet"
}

resource "oci_core_service_gateway" "service_gateway" {
```

```
  vcn_id   = oci_core_vcn.example_vcn.id
  services = ["All"]
}

resource "oci_core_private_endpoint" "example" {
  compartment_id       = "<compartment-id>"
  display_name         = "example-private-endpoint"
  subnet_id            = oci_core_subnet.example_subnet.id
  target_service_id    = "<target-service-ocid>"
  endpoint_fqdn        = "private.example.com"
}
```

IAM Permissions

plaintext

Allow group YourGroup to manage virtual-network-family in compartment YourCompartment

Allow group YourGroup to manage service-gateways in compartment YourCompartment

Allow group YourGroup to manage private-endpoints in compartment YourCompartment

Key Notes

- Replace placeholders like <subscription-id> <compartment-id> <target-service-ocid> and <backend-service-url> with your actual values.
- Ensure proper IAM roles or policies for users or service accounts managing these resources.
- For multi-cloud setups align the networking configurations and security rules across providers.

Project 6. Automated Networking Setup with Terraform

Automate the configuration of networking resources such as virtual networks peering connections VPNs NAT Gateways and VPCs across different cloud providers.

Automated Networking Setup with Terraform is a project aimed at simplifying the provisioning and configuration of networking resources across multiple cloud providers. Using Terraform's Infrastructure as Code (IaC) capabilities this project focuses on creating virtual networks establishing connectivity through VPNs and peering and configuring resources like NAT Gateways and VPCs. By automating these tasks the project ensures consistent scalable and reusable networking setups for AWS Azure GCP and Oracle Cloud.

AWS Networking Setup

1. **Install Terraform**
   Ensure Terraform is installed on your system and AWS CLI is configured with appropriate credentials.

Define the Provider

```
provider "aws" {

  region = "us-east-1"

}
```

Create VPC

```
resource "aws_vpc" "main" {

  cidr_block = "10.0.0.0/16"

  tags = {

    Name = "main-vpc"
```

```
  }
}
```

Configure Subnets

```
resource "aws_subnet" "public_subnet" {
  vpc_id            = aws_vpc.main.id
  cidr_block        = "10.0.1.0/24"
  map_public_ip_on_launch = true
  availability_zone = "us-east-1a"
}
```

Add Internet Gateway

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

Create NAT Gateway

```
resource "aws_nat_gateway" "nat" {
  subnet_id     = aws_subnet.public_subnet.id
  allocation_id = aws_eip.nat.id
}
```

Peering Connections (Optional)

```
resource "aws_vpc_peering_connection" "peer" {
  vpc_id        = aws_vpc.main.id
  peer_vpc_id   = "vpc-xxxxxx"
```

```
  peer_region   = "us-west-1"
}
```

Apply Changes

Run the commands

terraform init

terraform plan

terraform apply

Azure Networking Setup

Install Terraform

Install Terraform and configure Azure CLI with credentials.

Define the Provider

```
provider "azurerm" {
  features {}
}
```

Create Virtual Network (VNet)

```
resource "azurerm_virtual_network" "vnet" {
  name                = "myVNet"
  address_space       = ["10.0.0.0/16"]
  location            = "East US"
  resource_group_name = azurerm_resource_group.rg.name
```

```
}
```

Create Subnets

```
resource "azurerm_subnet" "subnet" {

  name                 = "mySubnet"

  resource_group_name  = azurerm_resource_group.rg.name

  virtual_network_name = azurerm_virtual_network.vnet.name

  address_prefixes     = ["10.0.1.0/24"]

}
```

Setup VPN Gateway

```
resource "azurerm_virtual_network_gateway" "vpn_gateway" {

  name                = "vpnGateway1"

  location            = azurerm_resource_group.rg.location

  resource_group_name   = azurerm_resource_group.rg.name

  type                = "Vpn"

  vpn_type              = "RouteBased"

}
```

Apply Changes

Run the commands

terraform init

terraform plan

terraform apply

GCP Networking Setup

1. **Install Terraform**
   Install Terraform and authenticate with GCP using the gcloud CLI.

Define the Provider

```
provider "google" {

  project = "my-project-id"

  region  = "us-central1"

}
```

Create VPC Network

```
resource "google_compute_network" "vpc" {

  name = "my-vpc"

}
```

Add Subnets

```
resource "google_compute_subnetwork" "subnet" {

  name          = "my-subnet"

  ip_cidr_range = "10.0.1.0/24"

  network       = google_compute_network.vpc.id

  region        = "us-central1"

}
```

Add VPN Gateway

```
resource "google_compute_vpn_gateway" "vpn_gateway" {
```

```
  name    = "vpn-gateway"
  network = google_compute_network.vpc.id
  region  = "us-central1"
}
```

Apply Changes

Run the commands
terraform init

terraform plan

terraform apply

Oracle Cloud Infrastructure (OCI) Networking Setup

1. **Install Terraform**
   Install Terraform and configure OCI CLI with API credentials.

Define the Provider

```
provider "oci" {
  tenancy_ocid     = "ocid1.tenancy.oc1..xxxxx"
  user_ocid        = "ocid1.user.oc1..xxxxx"
  fingerprint      = "your_fingerprint"
  private_key_path = "/path/to/your/private/key.pem"
  region           = "us-ashburn-1"
}
```

Create VCN (Virtual Cloud Network)

```
resource "oci_core_vcn" "vcn" {
  cidr_block = "10.0.0.0/16"
  display_name = "MyVCN"
  compartment_id = var.compartment_id
}
```

Create Subnets

```
resource "oci_core_subnet" "subnet" {
  vcn_id = oci_core_vcn.vcn.id
  cidr_block = "10.0.1.0/24"
  display_name = "MySubnet"
  compartment_id = var.compartment_id
}
```

Add Internet Gateway

```
resource "oci_core_internet_gateway" "ig" {
  compartment_id = var.compartment_id
  vcn_id         = oci_core_vcn.vcn.id
  display_name   = "MyIG"
  is_enabled     = true
}
```

Apply Changes

Run the commands
terraform init

terraform plan

terraform apply

---

2. High Availability & Disaster Recovery

Project 1. High Availability Architecture

High Availability (HA) architecture is designed to ensure that services and applications remain operational and accessible even in the case of failures ensuring minimal downtime. This project will help you deploy a highly available infrastructure across different cloud platforms (AWS Azure GCP and Oracle Cloud) using Terraform.

The goal is to provision resources that support fault tolerance and redundancy by utilizing load balancers multiple availability zones and auto-scaling groups.

---

1. AWS High Availability Architecture with Terraform

Key Components

- **VPC** Virtual Private Cloud for network isolation.
- **Subnets** Public and private subnets across multiple availability zones.
- **Load Balancer** Elastic Load Balancer (ELB) for distributing traffic.
- **Auto Scaling Group** Auto Scaling to ensure enough instances for availability.
- **EC2 Instances** Virtual machines to run applications.

Steps

1. **Install Terraform** Ensure you have Terraform installed and configured with AWS credentials.

Create VPC Define the VPC with CIDR blocks and subnets across different Availability Zones.

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}
```

Create Subnets

Define both public and private subnets in different Availability Zones.

```
resource "aws_subnet" "public" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.1.0/24"
  availability_zone = "us-east-1a"
  map_public_ip_on_launch = true
}
```

Set Up Security Groups

Configure security groups to allow inbound HTTP and SSH traffic.

Create Load Balancer

Define an Application Load Balancer (ALB) to distribute traffic across instances.

```
resource "aws_lb" "main" {
  name               = "main-alb"
  internal           = false
  load_balancer_type = "application"
  security_groups    = [aws_security_group.lb.id]
```

```
  subnets            = [aws_subnet.public.id]
}
```

Auto Scaling Group Create an Auto Scaling Group with a launch configuration and set the desired capacity.

```
resource "aws_autoscaling_group" "main" {
  desired_capacity     = 3
  max_size             = 5
  min_size             = 2
  vpc_zone_identifier  = [aws_subnet.public.id]
  launch_configuration = aws_launch_configuration.main.id
}
```

---

2. Azure High Availability Architecture with Terraform

Key Components

- **Virtual Network** A network for connecting resources.
- **Availability Sets** Distribute VMs across fault and update domains.
- **Load Balancer** Azure Load Balancer to distribute traffic.
- **VMs** Virtual machines running your applications.

Steps

Install Terraform Ensure you have Terraform installed and configured with Azure credentials.

Create Resource Group and Virtual Network

Define a resource group and virtual network.

```
resource "azurerm_resource_group" "main" {
  name     = "main-resource-group"
```

```
  location = "East US"
}

resource "azurerm_virtual_network" "main" {
  name                = "main-vnet"
  location            = azurerm_resource_group.main.location
  resource_group_name = azurerm_resource_group.main.name
  address_space       = ["10.0.0.0/16"]
}
```

Create Availability Set

Configure availability sets to ensure VMs are distributed.

```
resource "azurerm_availability_set" "main" {
  name                = "main-availability-set"
  location            = azurerm_resource_group.main.location
  resource_group_name = azurerm_resource_group.main.name
}
```

Create Load Balancer

Create an Azure Load Balancer for distributing traffic.

```
resource "azurerm_lb" "main" {
  name                = "main-lb"
  location            = azurerm_resource_group.main.location
```

```
  resource_group_name = azurerm_resource_group.main.name
}
```

Provision Virtual Machines

Create virtual machines in the availability set.

```
resource "azurerm_linux_virtual_machine" "main" {
  name                = "main-vm"
  resource_group_name = azurerm_resource_group.main.name
  location            = azurerm_resource_group.main.location
  size                = "Standard_DS1_v2"
  availability_set_id = azurerm_availability_set.main.id
}
```

---

3. GCP High Availability Architecture with Terraform

Key Components

- **VPC Network** A virtual private network for isolation.
- **Subnets** Distribute subnets across multiple regions.
- **Load Balancer** Global Load Balancer for distributing traffic.
- **Instances** VM instances running applications.

Steps

Install Terraform Ensure Terraform is configured with GCP credentials.

Create VPC Network Define a VPC network with subnets in multiple regions.

```
resource "google_compute_network" "main" {
  name                    = "main-network"
  auto_create_subnetworks = "true"
}
```

Create Load Balancer

Define a global HTTP(S) load balancer.

```
resource "google_compute_global_forwarding_rule" "main" {
  name       = "main-forwarding-rule"
  target     = google_compute_target_http_proxy.main.self_link
  port_range  = "80"
}
```

Create VM Instances

Provision VM instances in different regions for high availability.

```
resource "google_compute_instance" "main" {
  name         = "main-instance"
  machine_type = "e2-medium"
  zone         = "us-central1-a"
  network_interface {
    network = google_compute_network.main.name
    access_config {
      // External IP
```

```
  }
 }
}
```

---

4. Oracle Cloud High Availability Architecture with Terraform

Key Components

- **Virtual Cloud Network (VCN)** A network for isolation.
- **Availability Domains** Distribute resources across multiple availability domains.
- **Load Balancer** Oracle Cloud Load Balancer.
- **Instances** Compute instances for your applications.

Steps

- **Install Terraform** Make sure Terraform is installed and Oracle Cloud credentials are configured.

Create VCN

Define a VCN with subnets across different availability domains.

```
resource "oci_core_virtual_network" "main" {
 compartment_id = var.compartment_id
 cidr_block     = "10.0.0.0/16"
 display_name   = "main-vcn"
}
```

Set Up Load Balancer Define the load balancer for distributing incoming traffic.

```
resource "oci_load_balancer_load_balancer" "main" {
```

```
  compartment_id = var.compartment_id
  display_name   = "main-load-balancer"
  shape          = "100Mbps"
}
```

Create Instances Provision compute instances across multiple availability domains.

```
resource "oci_core_instance" "main" {
  compartment_id = var.compartment_id
  availability_domain = "UocmPHX-AD-1"
  shape              = "VM.Standard2.1"
  display_name       = "main-instance"
}
```

Project 2. Disaster Recovery Setup

Disaster Recovery (DR) refers to strategies and procedures designed to ensure the recovery of critical IT systems and data after a disaster or unexpected outage. In this Terraform project you will implement a disaster recovery setup across various cloud platforms (AWS Azure GCP Oracle) by leveraging infrastructure-as-code (IaC) principles. This will involve setting up backup systems failover mechanisms and other disaster recovery services to ensure business continuity.

The Terraform code will define the infrastructure and configurations required for disaster recovery across different cloud providers. Each platform will have unique resources and services but the overall objective will remain the same—protect your data and ensure minimal downtime in the event of a disaster.

Step-by-Step Guide for Disaster Recovery Setup

AWS Disaster Recovery Setup (Using Terraform)

1. **Set up AWS Provider**

- Define the AWS provider with necessary credentials and region.

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

2. **Backup Resources (e.g. S3 Buckets for Backup Storage)**
   - **Create S3 buckets to store backups.**

```
resource "aws_s3_bucket" "backup_bucket" {
  bucket = "disaster-recovery-backup"
  acl    = "private"
}
```

3. **Create EC2 Instances for Failover (if needed)**
   - Define EC2 instances that will serve as failover nodes.

```
resource "aws_instance" "failover_instance" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"
}
```

4. **Automate Backup of Critical Data**
   - Use AWS services like AWS Backup or EC2 snapshots to automate regular backups.
   - Example for EC2 snapshot

```
resource "aws_ec2_snapshot" "instance_snapshot" {
  volume_id = aws_instance.failover_instance.root_block_device[0].volume_id
  description = "Snapshot for disaster recovery"
}
```

5. **Set up RDS for DR**
    - Create a Multi-AZ RDS instance for database replication and failover.

```
resource "aws_db_instance" "primary_db" {
  engine = "mysql"
  instance_class = "db.t3.micro"
  multi_az = true
}
```

Azure Disaster Recovery Setup (Using Terraform)

1. **Set up Azure Provider**
    - Define the Azure provider with necessary credentials and region.

```
provider "azurerm" {
  features {}
}
```

2. **Backup Resources (e.g. Azure Blob Storage for Backups)**
    - Create a storage account for backups.

```
resource "azurerm_storage_account" "backup_storage" {
```

```
  name                = "backupstorageaccount"
  resource_group_name = azurerm_resource_group.main.name
  location            = "East US"
  account_tier        = "Standard"
  account_replication_type = "LRS"
}
```

3. **Failover with Virtual Machines**
    - Define Virtual Machines that will be used as failover resources.

```
resource "azurerm_virtual_machine" "failover_vm" {
  name                = "failover-vm"
  location            = "East US"
  resource_group_name = azurerm_resource_group.main.name
  size                = "Standard_B1ls"
  network_interface_ids = [azurerm_network_interface.failover_nic.id]
}
```

4. **Automated Backup of Virtual Machines**
    - Use Azure Backup for VM backup automation.

```
resource "azurerm_backup_protected_vm" "example" {
  resource_group_name = azurerm_resource_group.main.name
  vault_name          = azurerm_backup_vault.main.name
  virtual_machine_id  = azurerm_virtual_machine.failover_vm.id
}
```

5. **Set up Azure Database for DR**
    - Use Azure Database for MySQL with geo-replication enabled.

```
resource "azurerm_mysql_server" "primary_db" {
  name                = "mysql-primary-db"
  location            = "East US"
  resource_group_name = azurerm_resource_group.main.name
  version             = "5.7"
  ssl_enforcement     = "Enabled"
  storage_mb          = 51200
  backup_retention_days = 7
  geo_redundant_backup_enabled = true
}
```

GCP Disaster Recovery Setup (Using Terraform)

1. **Set up GCP Provider**
    - Define the GCP provider with necessary credentials and region.

```
provider "google" {
  project = "your-project-id"
  region  = "us-central1"
}
```

2. **Backup Resources (e.g. Google Cloud Storage Buckets)**
    - Create a Cloud Storage bucket for backups.

```
resource "google_storage_bucket" "backup_bucket" {
  name     = "disaster-recovery-backup"
  location = "US"
}
```

3. **Create VM Instances for Failover**
    - Define Google Compute Engine instances for failover.

```
resource "google_compute_instance" "failover_instance" {
  name         = "failover-instance"
  machine_type = "f1-micro"
  zone         = "us-central1-a"
  boot_disk {
    initialize_params {
      image = "projects/debian-cloud/global/images/family/debian-10"
    }
  }
}
```

4. **Set up Automated Backup with GCP Backup and DR**
    - Automate VM backups with GCP's snapshot or backup services.

```
resource "google_compute_disk" "snapshot_disk" {
```

```
  name  = "failover-instance-disk"
  type  = "pd-standard"
  zone  = "us-central1-a"
  size  = 50
}
```

5. **Database for DR (e.g. Cloud SQL for MySQL)**
    - Set up Cloud SQL for MySQL with automated backups and failover.

```
resource "google_sql_database_instance" "primary_db" {
  name             = "primary-db-instance"
  region           = "us-central1"
  database_version = "MYSQL_5_7"
  settings {
    tier = "db-f1-micro"
    backup_configuration {
      enabled = true
      start_time = "0300"
    }
  }
}
```

Oracle Disaster Recovery Setup (Using Terraform)

1. **Set up Oracle Cloud Provider**
    - Define the Oracle Cloud provider with credentials and region.

```
provider "oci" {
  tenancy_ocid     = "your-tenancy-ocid"
  user_ocid        = "your-user-ocid"
  fingerprint      = "your-fingerprint"
  private_key_path = "path-to-your-private-key"
  region           = "us-phoenix-1"
}
```

2. **Backup Resources (e.g. Oracle Cloud Object Storage for Backups)**
    - Create an object storage bucket for backups.

```
resource "oci_objectstorage_bucket" "backup_bucket" {
  compartment_id = "your-compartment-id"
  name           = "disaster-recovery-backup"
  storage_tier   = "Standard"
}
```

3. **Create Compute Instances for Failover**
    - Define Oracle Compute instances that will act as failover VMs.

```
resource "oci_core_instance" "failover_instance" {
  compartment_id = "your-compartment-id"
  availability_domain = "your-availability-domain"
  shape = "VM.Standard2.1"
  display_name = "failover-instance"
```

```
  create_vnic_details {
    subnet_id = "your-subnet-id"
  }
}
```

4. **Automate Backup with Oracle Cloud Backup**
    - Use Oracle Cloud Infrastructure Backup service for automated backup.

```
resource "oci_storage_backup" "instance_backup" {
  instance_id  = oci_core_instance.failover_instance.id
  backup_type  = "full"
  backup_time  = "0000"
}
```

5. **Database for DR (e.g. Oracle Autonomous Database)**
    - Set up an Autonomous Database with automated backup and DR capabilities.

```
resource "oci_database_autonomous_database" "primary_db" {
  compartment_id = "your-compartment-id"
  db_name        = "primary-db"
  cpu_core_count = 1
  data_storage_size_in_tbs = 1
  backup_config {
    backup_destination = "cloud"
    retention_period_in_days = 7
  }
```

}

Conclusion

This Terraform-based disaster recovery setup ensures that your infrastructure is resilient to outages across AWS Azure GCP and Oracle Cloud. The project includes the creation of backup resources failover mechanisms automated backups and database replication. Each cloud provider offers unique services and using Terraform we can manage them consistently across different environments.

Project 3 Disaster Recovery Simulation

Disaster Recovery (DR) simulation is a critical part of ensuring that your infrastructure is resilient to failures. The goal of this Terraform project is to simulate the recovery process in case of a disaster by setting up disaster recovery mechanisms across multiple cloud providers AWS Azure GCP and Oracle Cloud Infrastructure (OCI).

In this project we'll simulate disaster recovery by creating resources in one cloud provider (primary region) and configuring backup and restore strategies in another cloud provider (secondary region). This ensures that in the event of a failure you can recover from a different provider.

AWS

1. Disaster Recovery on AWS

In AWS we will create EC2 instances an S3 bucket and a VPC in one region (primary) and simulate disaster recovery by replicating these resources to another region (secondary).

Step-by-Step Setup

1. **Create a primary region infrastructure**
    - **VPC** Create a VPC in the primary AWS region.
    - **EC2 Instances** Create EC2 instances in this VPC.
    - **S3 Bucket** Set up an S3 bucket for backups.

```
provider "aws" {
  region = "us-east-1"  # Primary region
}

resource "aws_vpc" "primary_vpc" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_instance" "primary_instance" {
  ami           = "ami-0c55b159cbfafe1f0"  # Example AMI for EC2 instance
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.primary_subnet.id
}

resource "aws_s3_bucket" "primary_s3" {
  bucket = "my-backup-bucket-primary"
}
```

2. **Create disaster recovery infrastructure in secondary region**
    - **VPC** Create a VPC in the secondary AWS region.
    - **EC2 Instances** Create EC2 instances in the secondary region.
    - **S3 Bucket** Set up an S3 bucket in the secondary region for backup restoration.

```
provider "aws" {
  alias  = "secondary"
```

```
  region = "us-west-2"  # Secondary region
}
resource "aws_vpc" "secondary_vpc" {
  provider = aws.secondary
  cidr_block = "10.1.0.0/16"
}
resource "aws_instance" "secondary_instance" {
  provider     = aws.secondary
  ami          = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  subnet_id    = aws_subnet.secondary_subnet.id
}
resource "aws_s3_bucket" "secondary_s3" {
  provider = aws.secondary
  bucket   = "my-backup-bucket-secondary"
}
```

3. **Set up replication for disaster recovery**
    - Use AWS S3 Cross-Region Replication to back up data from the primary S3 bucket to the secondary region's bucket.

```
resource "aws_s3_bucket_object" "replication_object" {
  bucket = aws_s3_bucket.primary_s3.bucket
  key    = "backup-data.zip"
  source = "data/backup.zip"
```

```
}


resource "aws_s3_bucket_replication_configuration" "replication" {
  role = "arnawsiam123456789012role/S3ReplicationRole"
  rules {
    status = "Enabled"
    prefix = ""
    destination {
      bucket = aws_s3_bucket.secondary_s3.bucket
    }
  }
}
```

Azure

2. Disaster Recovery on Azure

Introduction

In Azure we will use Virtual Machines (VMs) Virtual Networks and Storage accounts for disaster recovery. The primary setup will be in one region and we will replicate to a secondary region.

Step-by-Step Setup

1. **Create primary region infrastructure**
    - **Virtual Network** Set up a Virtual Network in the primary region.
    - **Virtual Machine** Deploy a Virtual Machine (VM) in the primary region.

- **Storage Account** Set up a storage account to hold backups.

Terraform Configuration

```
provider "azurerm" {
  features {}
}

resource "azurerm_virtual_network" "primary_vnet" {
  name                = "primary-vnet"
  address_space       = ["10.0.0.0/16"]
  location            = "East US"
  resource_group_name = azurerm_resource_group.primary_rg.name
}

resource "azurerm_linux_virtual_machine" "primary_vm" {
  name                = "primary-vm"
  resource_group_name = azurerm_resource_group.primary_rg.name
  location            = "East US"
  size                = "Standard_B1ms"
  network_interface_ids = [azurerm_network_interface.primary_nic.id]
  admin_username      = "adminuser"
  admin_password      = "Password123!"
}
```

```
resource "azurerm_storage_account" "primary_storage" {
  name                     = "primarystorage"
  resource_group_name      = azurerm_resource_group.primary_rg.name
  location                 = "East US"
  account_tier             = "Standard"
  account_replication_type = "LRS"
}
```

2. **Create disaster recovery infrastructure in secondary region**
    - **Virtual Network** Set up a Virtual Network in the secondary region.
    - **Virtual Machine** Deploy a VM in the secondary region.
    - **Storage Account** Set up a storage account in the secondary region.

```
resource "azurerm_virtual_network" "secondary_vnet" {
  name                = "secondary-vnet"
  address_space       = ["10.1.0.0/16"]
  location            = "West US"
  resource_group_name = azurerm_resource_group.secondary_rg.name
}
resource "azurerm_linux_virtual_machine" "secondary_vm" {
  name                = "secondary-vm"
  resource_group_name = azurerm_resource_group.secondary_rg.name
  location            = "West US"
  size                = "Standard_B1ms"
  network_interface_ids = [azurerm_network_interface.secondary_nic.id]
```

```
  admin_username      = "adminuser"
  admin_password      = "Password123!"
}

resource "azurerm_storage_account" "secondary_storage" {
  name                     = "secondarystorage"
  resource_group_name      = azurerm_resource_group.secondary_rg.name
  location                 = "West US"
  account_tier             = "Standard"
  account_replication_type = "LRS"
}
```

3. **Set up replication for disaster recovery**
    - Use Azure Blob Storage replication or Azure Site Recovery to replicate data and VMs to the secondary region.

GCP

3. Disaster Recovery on GCP

Introduction

In GCP we will create Compute Engine instances Cloud Storage and Virtual Private Cloud (VPC) networks in one region with replication to another region for disaster recovery.

Step-by-Step Setup

1. **Create primary region infrastructure**
    - **VPC** Set up a VPC in the primary region.
    - **Compute Engine** Deploy a Compute Engine instance in the primary region.
    - **Cloud Storage Bucket** Create a Cloud Storage bucket for backups.

```
provider "google" {
  project = "my-project"
  region  = "us-central1"  # Primary region
}
resource "google_compute_instance" "primary_instance" {
  name         = "primary-instance"
  machine_type = "e2-micro"
  zone         = "us-central1-a"
  boot_disk {
    initialize_params {
      image = "projects/debian-cloud/global/images/family/debian-10"
    }
  }
  network_interface {
    network = "default"
    access_config {}
  }
}
resource "google_storage_bucket" "primary_bucket" {
  name     = "my-backup-bucket-primary"
  location = "US"
}
```

2. **Create disaster recovery infrastructure in secondary region**

- **VPC** Set up a VPC in the secondary region.
- **Compute Engine** Create a Compute Engine instance in the secondary region.
- **Cloud Storage Bucket** Set up a Cloud Storage bucket for backups.

```
provider "google" {
  alias   = "secondary"
  project = "my-project"
  region  = "us-east1"  # Secondary region
}
resource "google_compute_instance" "secondary_instance" {
  provider    = google.secondary
  name        = "secondary-instance"
  machine_type = "e2-micro"
  zone        = "us-east1-b"
  boot_disk {
    initialize_params {
      image = "projects/debian-cloud/global/images/family/debian-10"
    }
  }
  network_interface {
    network = "default"
    access_config {}
  }
}
```

```
resource "google_storage_bucket" "secondary_bucket" {
  provider = google.secondary
  name     = "my-backup-bucket-secondary"
  location = "US"
}
```

3. **Set up replication for disaster recovery**
    - Use **Google Cloud Storage Object Versioning** or **Persistent Disk Snapshotting** for backup and recovery.

Oracle

4. Disaster Recovery on Oracle Cloud Infrastructure (OCI)

Introduction

In OCI we will set up Compute instances VCN and Object Storage in one region and replicate to another region for disaster recovery.

Step-by-Step Setup

1. **Create primary region infrastructure**
    - **VCN** Set up a Virtual Cloud Network (VCN) in the primary region.
    - **Compute Instances** Deploy compute instances in the primary region.
    - **Object Storage** Create an Object Storage bucket for backup.

```
provider "oci" {
  region = "us-phoenix-1"  # Primary region
}
resource "oci_core_virtual_network" "primary_vcn" {
  cidr_block = "10.0.0.0/16"
```

```
  compartment_id = var.compartment_ocid
  display_name = "primary-vcn"
}
resource "oci_core_instance" "primary_instance" {
  compartment_id = var.compartment_ocid
  availability_domain = "UocmUS-ASHBURN-AD-1"
  shape = "VM.Standard.E2.1.Micro"
  display_name = "primary-instance"
  subnet_id = oci_core_subnet.primary_subnet.id
}
resource "oci_objectstorage_bucket" "primary_bucket" {
  compartment_id = var.compartment_ocid
  namespace      = var.namespace
  name           = "primary-backup-bucket"
}
```

2. **Create disaster recovery infrastructure in secondary region**
    - **VCN** Set up a Virtual Cloud Network (VCN) in the secondary region.
    - **Compute Instances** Deploy compute instances in the secondary region.
    - **Object Storage** Create an Object Storage bucket for backup.

```
provider "oci" {
  alias  = "secondary"
  region = "us-ashburn-1"  # Secondary region
}
resource "oci_core_virtual_network" "secondary_vcn" {
```

```
  provider = oci.secondary
  cidr_block = "10.1.0.0/16"
  compartment_id = var.compartment_ocid
  display_name = "secondary-vcn"
}
resource "oci_core_instance" "secondary_instance" {
  provider = oci.secondary
  compartment_id = var.compartment_ocid
  availability_domain = "UocmUS-ASHBURN-AD-2"
  shape = "VM.Standard.E2.1.Micro"
  display_name = "secondary-instance"
  subnet_id = oci_core_subnet.secondary_subnet.id
}
resource "oci_objectstorage_bucket" "secondary_bucket" {
  provider = oci.secondary
  compartment_id = var.compartment_ocid
  namespace      = var.namespace
  name           = "secondary-backup-bucket"
}
```

3. **Set up replication for disaster recovery**
    - Use **OCI Block Volume Replication** or **Object Storage Cross-Region Replication** for backup and recovery.

This disaster recovery simulation provides a multi-cloud approach to ensure that if one cloud provider or region goes down critical infrastructure can be restored quickly in another region or provider.

Project 4. Disaster Recovery and High Availability Setup

Automate the creation of highly available infrastructure with multi-region setups load balancing failover mechanisms and data replication for fault tolerance.

Disaster Recovery and High Availability Setup Project

Introduction

This project focuses on automating the creation of a highly available infrastructure with disaster recovery capabilities using Terraform. The goal is to ensure your infrastructure is resilient to failures by setting up multi-region deployments load balancing failover mechanisms and data replication across different cloud providers. This approach will help you ensure fault tolerance and high availability for your applications in AWS Azure GCP and Oracle Cloud.

---

1. AWS Setup

Overview

For AWS we will use services such as **Amazon EC2 Amazon RDS Elastic Load Balancer (ELB) Route 53** and **Amazon S3** to implement multi-region high availability with failover and data replication.

Steps

1. **Setup EC2 Instances**
    - Define EC2 instances in multiple availability zones (AZs) in different regions for failover capabilities.

Example

```
resource "aws_instance" "web" {

  ami           = "ami-xxxxxxxx"

  instance_type = "t2.micro"

  availability_zone = "us-east-1a"
```

}

2. **Setup Load Balancer**
    - Create an Application Load Balancer (ALB) to distribute traffic across EC2 instances.

Example
```
resource "aws_lb" "app_lb" {

  name               = "app-lb"

  internal           = false

  load_balancer_type = "application"

  security_groups    = [aws_security_group.sg.id]

  subnets            = [aws_subnet.subnet.id]

}
```

3. **Data Replication with Amazon RDS**
    - Create RDS instances in multiple regions for cross-region replication.

Example
```
resource "aws_db_instance" "primary" {

  engine         = "mysql"

  instance_class = "db.m5.large"

  allocated_storage = 20

  multi_az       = true

  storage_type   = "gp2"

}
```

4. **Route 53 for DNS Failover**
    - Use AWS Route 53 to manage DNS and automatic failover between regions.

Example

```
resource "aws_route53_record" "failover_record" {

  zone_id = aws_route53_zone.primary.id

  name    = "myapp.example.com"

  type    = "A"

  alias {

    name                   = aws_lb.app_lb.dns_name

    zone_id                = aws_lb.app_lb.zone_id

    evaluate_target_health = true

  }

}
```

5. **S3 for Backup and Replication**
    - Set up S3 buckets for backup with cross-region replication enabled.

Example

```
resource "aws_s3_bucket" "bucket" {

  bucket = "my-backup-bucket"

  region = "us-east-1"

}



resource "aws_s3_bucket_object" "object" {

  bucket = aws_s3_bucket.bucket.id

  key    = "backup-data"

  source = "data.zip"

}
```

---

2. Azure Setup

Overview

Azure offers services such as **Azure VMs Azure Load Balancer Azure SQL Database** and **Azure Blob Storage** to implement disaster recovery and high availability.

Steps

1. **Setup Virtual Machines**
    - Create Azure VMs across multiple availability zones for high availability.

Example

```
resource "azurerm_linux_virtual_machine" "vm" {

  name                = "my-vm"

  resource_group_name = azurerm_resource_group.rg.name

  location            = "East US"

  size                = "Standard_DS1_v2"

  availability_zone   = 1

}
```

2. **Setup Load Balancer**
    - Use Azure Load Balancer to distribute traffic across VMs.

Example

```
resource "azurerm_lb" "load_balancer" {

  name                = "my-load-balancer"

  location            = "East US"

  resource_group_name = azurerm_resource_group.rg.name

}
```

3. **SQL Database Replication**
    - Set up SQL Database with geo-replication enabled for disaster recovery.

Example

```
resource "azurerm_sql_server" "primary" {

  name                = "primary-sql-server"

  resource_group_name = azurerm_resource_group.rg.name

  location            = "East US"

  version             = "12.0"

}
```

4. **Blob Storage for Backup**
    - Set up Azure Blob Storage for backups with replication.

Example

```
resource "azurerm_storage_account" "backup" {

  name                = "mybackupstorage"

  resource_group_name = azurerm_resource_group.rg.name

  location            = "East US"

  account_tier        = "Standard"

  account_replication_type = "LRS"

}
```

---

3. GCP Setup

Overview

Google Cloud uses services like **Compute Engine Google Cloud Load Balancing Cloud SQL** and **Cloud Storage** for high availability setups.

Steps

1. **Setup Compute Instances**

- Deploy instances in multiple zones for high availability.

Example

```
resource "google_compute_instance" "default" {
  name         = "my-instance"
  machine_type = "n1-standard-1"
  zone         = "us-central1-a"
  boot_disk {
    initialize_params {
      image = "debian-9-stretch-v20201014"
    }
  }
}
```

2. **Setup Global Load Balancer**
   - Use GCP's Global Load Balancer to distribute traffic across multiple zones.

Example

```
resource "google_compute_forwarding_rule" "default" {
  name       = "http-lb-rule"
  target     = google_compute_target_http_proxy.default.id
  port_range  = "80"
  ip_address  = google_compute_address.default.address
}
```

3. **Cloud SQL for Database Replication**

   Use Cloud SQL with replication enabled for high availability.

Example

```
resource "google_sql_database_instance" "primary" {
  name = "primary-instance"
  region = "us-central1"
  database_version = "MYSQL_5_7"
  settings {
    tier = "db-n1-standard-1"
    availability_type = "REGIONAL"
  }
}
```

4. **Cloud Storage for Backup**
    - Enable Cloud Storage with object versioning for backup purposes.

Example

```
resource "google_storage_bucket" "backup" {
  name          = "my-backup-bucket"
  location      = "US"
  versioning {
    enabled = true
  }
}
```

    -

---

4. Oracle Cloud Setup

Overview

Oracle Cloud provides **Compute Instances Load Balancer Oracle Autonomous Database** and **Object Storage** to set up highly available and fault-tolerant systems.

Steps

1. **Setup Compute Instances**
    - Deploy instances in multiple availability domains for high availability.

Example

```
resource "oci_core_instance" "instance" {

  availability_domain = "UocmPHX-AD-1"

  compartment_id      = oci_identity_compartment.compartment.id

  shape               = "VM.Standard2.1"

  display_name        = "web-instance"

}
```

2. **Setup Load Balancer**
    - Use Oracle Cloud Load Balancer for distributing traffic.

Example

```
resource "oci_lb_load_balancer" "load_balancer" {

  compartment_id      = oci_identity_compartment.compartment.id

  display_name        = "my-load-balancer"

  shape               = "100Mbps"

  subnet_id           = oci_core_subnet.subnet.id

}
```

3. **Autonomous Database Replication**

- Set up Autonomous Database with Data Guard for replication and failover.

Example

```
resource "oci_database_autonomous_database" "adb" {
  compartment_id       = oci_identity_compartment.compartment.id
  db_name              = "mydb"
  cpu_core_count       = 1
  data_storage_size_in_tbs = 1
  db_workload          = "OLTP"
}
```

4. **Object Storage for Backup**
   - Use Oracle Object Storage for backups and data replication.

Example

```
resource "oci_objectstorage_bucket" "backup" {
  compartment_id = oci_identity_compartment.compartment.id
  display_name   = "my-backup-bucket"
  namespace      = "my-namespace"
}
```

Conclusion

Each cloud provider offers unique services to ensure high availability and disaster recovery. By using Terraform you can automate the setup and configuration of multi-region deployments load balancing failover and data replication for fault tolerance. This approach will help ensure that your applications are resilient to failures across AWS Azure GCP and Oracle Cloud.

Project 5. Disaster Recovery Setup

Build a disaster recovery plan using Terraform by replicating infrastructure across different regions or availability zones.

1st Example

Disaster Recovery Setup with Terraform Multi-Cloud Infrastructure Replication

Disaster Recovery (DR) plan is essential to ensure business continuity in case of failure. In this project we'll replicate infrastructure across multiple regions or availability zones to minimize downtime and ensure high availability. Terraform will be used to automate the replication and setup of resources on different cloud platforms including AWS Azure GCP and Oracle Cloud.

AWS Disaster Recovery Setup with Terraform

Prerequisites

- AWS account
- AWS CLI configured
- Terraform installed

Steps

Set Up AWS Provider
Define the AWS provider in the main.tf file to manage the infrastructure.

```
provider "aws" {

  region = "us-west-1"

}
```

Define Resources in Primary Region
Create an EC2 instance and an S3 bucket in the primary region (e.g. us-west-1).

```
resource "aws_instance" "primary_instance" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"
}

resource "aws_s3_bucket" "primary_bucket" {
  bucket = "primary-bucket"
}
```

Replicate Infrastructure to Secondary Region
Use the provider block for a secondary region (e.g. us-east-1) to replicate the resources.

```
provider "aws" {
  region = "us-east-1"
  alias  = "secondary"
}

resource "aws_instance" "secondary_instance" {
  provider      = aws.secondary
  ami           = "ami-12345678"
  instance_type = "t2.micro"
}
```

```
resource "aws_s3_bucket" "secondary_bucket" {
  provider = aws.secondary
  bucket   = "secondary-bucket"
}
```

Set Up S3 Bucket Replication
Configure S3 bucket replication between the primary and secondary regions.

```
resource "aws_s3_bucket_object" "replica_object" {
  bucket = aws_s3_bucket.primary_bucket.bucket
  key    = "replica_data"
  source = "data.txt"
}
```

Apply Configuration
Run Terraform to apply the configuration.

```
terraform init
terraform apply
```

Azure Disaster Recovery Setup with Terraform

Prerequisites

- Azure account
- Azure CLI configured

- Terraform installed

Steps

Set Up Azure Provider
Define the Azure provider in main.tf.

```
provider "azurerm" {
  features {}
}
```

Define Resources in Primary Region
Create a virtual network and an Azure VM in the primary region (e.g. East US).

```
resource "azurerm_virtual_network" "primary_vnet" {
  name                = "primary-vnet"
  address_space       = ["10.0.0.0/16"]
  location            = "East US"
  resource_group_name = "primary-rg"
}

resource "azurerm_virtual_machine" "primary_vm" {
  name                  = "primary-vm"
  resource_group_name   = "primary-rg"
  location              = "East US"
  vm_size               = "Standard_B1ls"
  network_interface_ids = [azurerm_network_interface.primary_nic.id]
```

}

Replicate Infrastructure to Secondary Region
Define resources in the secondary region (e.g. West US).

```
resource "azurerm_virtual_network" "secondary_vnet" {
  name                = "secondary-vnet"
  address_space       = ["10.1.0.0/16"]
  location            = "West US"
  resource_group_name = "secondary-rg"
}


resource "azurerm_virtual_machine" "secondary_vm" {
  name                  = "secondary-vm"
  resource_group_name   = "secondary-rg"
  location              = "West US"
  vm_size               = "Standard_B1ls"
  network_interface_ids = [azurerm_network_interface.secondary_nic.id]
}
```

Set Up Azure Backup for DR
Configure Azure backup services to replicate VM snapshots between regions.

```
resource "azurerm_backup_protected_vm" "primary_vm_backup" {
  resource_group_name     = "primary-rg"
  recovery_vault_name     = "primary-backup-vault"
  virtual_machine_id      = azurerm_virtual_machine.primary_vm.id
  backup_management_type = "AzureIaasVM"
```

}

Apply Configuration

Initialize and apply the configuration.

terraform init

terraform apply

GCP Disaster Recovery Setup with Terraform

Prerequisites

- Google Cloud account
- Google Cloud SDK configured
- Terraform installed

Steps

Set Up GCP Provider
Define the GCP provider.

```
provider "google" {
  project = "your-gcp-project-id"
  region  = "us-central1"
}
```

Define Resources in Primary Region

Create a Compute instance in the primary region.

```
resource "google_compute_instance" "primary_instance" {
```

```
  name         = "primary-instance"
  machine_type = "e2-micro"
  zone         = "us-central1-a"
  boot_disk {
    initialize_params {
      image = "debian-9-stretch-v20191210"
    }
  }
}
```

Replicate Infrastructure to Secondary Region

Define resources in the secondary region (e.g. us-west1).

```
provider "google" {
  region = "us-west1"
  alias  = "secondary"
}

resource "google_compute_instance" "secondary_instance" {
  provider     = google.secondary
  name         = "secondary-instance"
  machine_type = "e2-micro"
  zone         = "us-west1-a"
  boot_disk {
    initialize_params {
```

```
    image = "debian-9-stretch-v20191210"
  }
 }
}
```

Set Up GCP Storage Bucket Replication

Configure cross-region replication for Google Cloud Storage buckets.

```
resource "google_storage_bucket" "primary_bucket" {
 name     = "primary-bucket"
 location = "US-CENTRAL1"
}

resource "google_storage_bucket_object" "replica_object" {
 name   = "replica_object"
 bucket = google_storage_bucket.primary_bucket.name
 source = "data.txt"
}
```

Apply Configuration

Run Terraform to set up the infrastructure.

```
terraform init
terraform apply
```

Oracle Cloud Disaster Recovery Setup with Terraform

Prerequisites

- Oracle Cloud account
- Oracle CLI configured
- Terraform installed

Steps

Set Up Oracle Cloud Provider
Define the Oracle Cloud provider.

```
provider "oci" {
  tenancy_ocid     = "your-tenancy-ocid"
  user_ocid        = "your-user-ocid"
  fingerprint      = "your-fingerprint"
  private_key_path = "path/to/your/private-key"
  region           = "us-phoenix-1"
}
```

Define Resources in Primary Region

Create an instance in the primary region.

```
resource "oci_core_instance" "primary_instance" {
  availability_domain = "UocmPHX-AD-1"
  compartment_id      = "your-compartment-id"
  shape               = "VM.Standard2.1"
  display_name        = "primary-instance"
}
```

Replicate Infrastructure to Secondary Region
Define resources in the secondary region (e.g. us-ashburn-1).

```
provider "oci" {
  region = "us-ashburn-1"
  alias  = "secondary"
}
resource "oci_core_instance" "secondary_instance" {
  provider          = oci.secondary
  availability_domain = "UocmASH-AD-1"
  compartment_id     = "your-compartment-id"
  shape             = "VM.Standard2.1"
  display_name      = "secondary-instance"
}
```

Set Up Oracle Cloud Storage Replication

Configure the replication of storage volumes between regions.

```
resource "oci_objectstorage_bucket" "primary_bucket" {
  compartment_id = "your-compartment-id"
  name          = "primary-bucket"
  storage_tier   = "Standard"
}
resource "oci_objectstorage_object" "replica_object" {
  bucket_name    = oci_objectstorage_bucket.primary_bucket.name
  object_name    = "replica_object"
```

```
  source         = "data.txt"

}
```

Apply Configuration
Run Terraform to apply the infrastructure configuration.

```
terraform init

terraform apply
```

2nd Example

Disaster Recovery Setup with Terraform (Multi-Cloud)

Introduction

Disaster Recovery (DR) is a critical part of an organization's business continuity plan ensuring that in case of infrastructure failure critical services are restored as quickly as possible. Using Terraform we can automate the replication of cloud resources across multiple regions or even across different cloud providers to increase redundancy minimize downtime and improve fault tolerance.

This setup involves deploying infrastructure components such as Virtual Machines (VMs) Networking and Storage in different cloud regions or availability zones and replicating data to ensure that in the event of failure traffic can be rerouted and services will still be accessible.

AWS Disaster Recovery Setup with Terraform

Steps for Setup

Define AWS Providers for Multiple Regions
Configure multiple providers for different AWS regions to replicate infrastructure.

```
provider "aws" {
```

```
  region = "us-east-1"  # Primary region
}

provider "aws" {
  region = "us-west-2"  # Secondary region
  alias  = "secondary"
}
```

Set Up EC2 Instances in Primary and Secondary Regions

Create EC2 instances in both the primary and secondary regions.

```
resource "aws_instance" "primary_instance" {
  ami           = "ami-0abcdef1234567890"
  instance_type = "t2.micro"
}

resource "aws_instance" "secondary_instance" {
  provider      = aws.secondary
  ami           = "ami-0abcdef1234567890"
  instance_type = "t2.micro"
}
```

Set Up Load Balancer (Optional)

For failover configure an AWS Elastic Load Balancer (ELB) to distribute traffic across multiple regions.

```
resource "aws_lb" "dr_lb" {
  name               = "dr-lb"
  internal           = false
```

```
  load_balancer_type = "application"

  security_groups    = [aws_security_group.lb_sg.id]

  subnets            = [aws_subnet.primary_subnet.id aws_subnet.secondary_subnet.id]

}
```

Set Up S3 Bucket Replication for Data Backup

Use S3 bucket replication to back up data across regions.

```
resource "aws_s3_bucket" "primary_bucket" {

  bucket = "primary-bucket"

}


resource "aws_s3_bucket_replication_configuration" "replication" {

  role = aws_iam_role.replication_role.arn

  rules {

    status = "Enabled"

    destination {

      bucket = "arnawss3secondary-bucket"

    }

  }

}
```

Terraform Commands

Initialize Terraform and apply the configuration.

```
terraform init

terraform apply
```

Azure Disaster Recovery Setup with Terraform

Steps for Setup

Define Azure Providers for Multiple Regions
Configure the provider for both regions (e.g. East US and West US).

```
provider "azurerm" {
  features {}
  region = "East US"
}

provider "azurerm" {
  features {}
  region = "West US"
  alias  = "secondary"
}
```

Create Azure Virtual Machines in Both Regions
Define VM resources for both regions.

```
resource "azurerm_virtual_machine" "primary_vm" {
  name                = "primary-vm"
  resource_group_name   = "primary-rg"
  location            = "East US"
```

```
  size                = "Standard_B1ms"
  network_interface_ids = [azurerm_network_interface.primary_nic.id]
}

resource "azurerm_virtual_machine" "secondary_vm" {
  provider            = azurerm.secondary
  name                = "secondary-vm"
  resource_group_name   = "secondary-rg"
  location            = "West US"
  size                = "Standard_B1ms"
  network_interface_ids = [azurerm_network_interface.secondary_nic.id]
}
```

Set Up Azure Storage Replication
Use Azure Storage replication to copy data across regions.

```
resource "azurerm_storage_account" "primary_storage" {
  name                     = "primarystorage"
  resource_group_name      = "primary-rg"
  location                 = "East US"
  account_tier             = "Standard"
  account_replication_type = "GRS"  # Geo-redundant storage
}
```

Set Up Azure Load Balancer for Failover
If desired create an Azure Load Balancer to distribute traffic.

```
resource "azurerm_lb" "dr_lb" {

  name                = "dr-lb"

  location            = "East US"

  resource_group_name = "primary-rg"

}
```

Terraform Commands

Run Terraform commands to initialize and apply.

```
terraform init

terraform apply
```

GCP Disaster Recovery Setup with Terraform

Steps for Setup

Configure GCP Providers for Multiple Regions

Set up GCP providers for primary and secondary regions.

```
provider "google" {

  project = "your-project-id"

  region  = "us-central1"

}


provider "google" {
```

```
  project = "your-project-id"
  region  = "us-west1"
  alias   = "secondary"
}
```

Create Virtual Machines in Primary and Secondary Regions

Define compute instances in both regions.

```
resource "google_compute_instance" "primary_instance" {
  name         = "primary-instance"
  machine_type = "e2-micro"
  zone         = "us-central1-a"
  boot_disk {
    initialize_params {
      image = "debian-9-stretch-v20191210"
    }
  }
}
resource "google_compute_instance" "secondary_instance" {
  provider     = google.secondary
  name         = "secondary-instance"
  machine_type = "e2-micro"
  zone         = "us-west1-a"
  boot_disk {
```

```
    initialize_params {
      image = "debian-9-stretch-v20191210"
    }
  }
}
```

Set Up Cloud Storage Bucket Replication
Replicate storage across regions for data redundancy.

```
resource "google_storage_bucket" "primary_bucket" {
  name     = "primary-bucket"
  location = "US-CENTRAL1"
}
resource "google_storage_bucket" "secondary_bucket" {
  provider = google.secondary
  name     = "secondary-bucket"
  location = "US-WEST1"
}
```

Terraform Commands

Initialize and apply configuration.
terraform init

terraform apply

Oracle Cloud Disaster Recovery Setup with Terraform

Steps for Setup

Define Oracle Cloud Provider
Set up Oracle Cloud provider configuration.

```
provider "oci" {
  tenancy_ocid     = "your-tenancy-ocid"
  user_ocid        = "your-user-ocid"
  fingerprint      = "your-fingerprint"
  private_key_path = "path/to/private-key"
  region           = "us-phoenix-1"
}

provider "oci" {
  tenancy_ocid     = "your-tenancy-ocid"
  user_ocid        = "your-user-ocid"
  fingerprint      = "your-fingerprint"
  private_key_path = "path/to/private-key"
  region           = "us-ashburn-1"
  alias            = "secondary"
}
```

Create Oracle Cloud Instances in Primary and Secondary Regions

Define compute instances in both regions.

```
resource "oci_core_instance" "primary_instance" {
  availability_domain = "UocmPHX-AD-1"
```

```
  compartment_id     = "your-compartment-id"
  shape              = "VM.Standard2.1"
  display_name       = "primary-instance"
}

resource "oci_core_instance" "secondary_instance" {
  provider           = oci.secondary
  availability_domain = "UocmASH-AD-1"
  compartment_id     = "your-compartment-id"
  shape              = "VM.Standard2.1"
  display_name       = "secondary-instance"
}
```

Set Up Object Storage Replication
Use Oracle Cloud's object storage for data replication across regions.

```
resource "oci_objectstorage_bucket" "primary_bucket" {
  compartment_id = "your-compartment-id"
  name           = "primary-bucket"
  storage_tier   = "Standard"
}

resource "oci_objectstorage_bucket" "secondary_bucket" {
  provider      = oci.secondary
```

```
  compartment_id = "your-compartment-id"
  name           = "secondary-bucket"
  storage_tier   = "Standard"
}
```

Terraform Commands
Run Terraform to initialize and apply the plan.

terraform init

terraform apply

Conclusion

Setting up disaster recovery (DR) using Terraform across multiple cloud providers helps organizations ensure high availability fault tolerance and business continuity. By automating the replication of infrastructure storage and data you can minimize downtime and quickly recover from regional outages or failures.

Adjust each of the configurations above based on your organization's specific requirements such as VM types storage options and regions

---

Project 6 Automated Disaster Recovery for Cloud Infrastructure

Use Terraform to design a disaster recovery plan automating the failover process between multiple regions or cloud providers for high availability.

This Terraform project aims to automate the disaster recovery process for cloud infrastructure ensuring high availability by using multiple regions or cloud providers. By leveraging Terraform you can define your infrastructure as code and create automated processes for failover between regions or even across different cloud platforms like AWS Azure GCP and Oracle.

Objective

The main goal of this project is to create an infrastructure that automatically recovers from failures by triggering failover to a secondary region or cloud provider. This ensures that your applications remain available even in the event of a disaster or outage.

---

1. AWS

Prerequisites

- AWS CLI configured with proper credentials.
- Terraform installed.
- Two AWS regions (Primary and Secondary) for failover.

Terraform Configuration

1. **Define Providers**

```
provider "aws" {
  region = "us-west-2"  # Primary region
}

provider "aws" {
  alias  = "secondary"
  region = "us-east-1"  # Secondary region
}
```

2. **Define Resources (e.g. EC2 Instances S3 Buckets)**

```
resource "aws_instance" "primary" {
  ami           = "ami-xxxxxxxxxx"
  instance_type = "t2.micro"
  region        = "us-west-2"
}
resource "aws_instance" "secondary" {
  ami           = "ami-xxxxxxxxxx"
```

```
  instance_type = "t2.micro"
  region        = "us-east-1"
  provider      = aws.secondary
}
```

3. **Failover Mechanism with AWS Route 53 for DNS Failover**

```
resource "aws_route53_record" "primary" {
  zone_id = "YOUR_ZONE_ID"
  name    = "example.com"
  type    = "A"
  ttl     = 60
  records = [aws_instance.primary.public_ip]
}
resource "aws_route53_record" "secondary" {
  zone_id = "YOUR_ZONE_ID"
  name    = "example.com"
  type    = "A"
  ttl     = 60
  records = [aws_instance.secondary.public_ip]
  provider = aws.secondary
}
```

4. **Automatic Failover via Health Checks**

```
resource "aws_route53_health_check" "primary" {
```

```
  fqdn              = "example.com"
  type              = "HTTP"
  resource_path     = "/"
  failure_threshold = 3
  port              = 80
  child_health_checks = []
}
resource "aws_route53_record" "failover" {
  zone_id = "YOUR_ZONE_ID"
  name    = "example.com"
  type    = "A"
  ttl     = 60
  records = [aws_instance.primary.public_ip]
  set_identifier = "primary"
  health_check_id = aws_route53_health_check.primary.id
}
resource "aws_route53_record" "secondary_failover" {
  zone_id = "YOUR_ZONE_ID"
  name    = "example.com"
  type    = "A"
  ttl     = 60
  records = [aws_instance.secondary.public_ip]
  set_identifier = "secondary"
```

```
  health_check_id = aws_route53_health_check.primary.id
  provider = aws.secondary
}
```

---

2. Azure

Prerequisites

- Azure CLI configured with proper credentials.
- Terraform installed.
- Two Azure regions (Primary and Secondary).

Terraform Configuration

1. **Define Providers**

```
provider "azurerm" {
  features = {}
  region   = "East US"  # Primary region
}
provider "azurerm" {
  alias    = "secondary"
  features = {}
  region   = "West US"  # Secondary region
}
```

2. **Define Resources (e.g. Virtual Machines Storage Accounts)**

```
resource "azurerm_linux_virtual_machine" "primary" {
  name                = "primary-vm"
```

```
  resource_group_name = "primary-rg"
  location            = "East US"
  size                = "Standard_B1s"
}
resource "azurerm_linux_virtual_machine" "secondary" {
  name                = "secondary-vm"
  resource_group_name = "secondary-rg"
  location            = "West US"
  size                = "Standard_B1s"
  provider            = azurerm.secondary
}
```

3. **Failover Mechanism using Azure Traffic Manager**

```
resource "azurerm_traffic_manager_profile" "failover" {
  name                = "traffic-manager-profile"
  resource_group_name = "rg"
  traffic_routing_method = "Failover"
  relative_priority = 1

  monitor {
    protocol = "HTTP"
    port     = 80
    path     = "/"
  }
```

```
  endpoint {
    name                 = "primary"
    resource_id          = azurerm_linux_virtual_machine.primary.id
    endpoint_location    = "East US"
    priority             = 1
  }
  endpoint {
    name                 = "secondary"
    resource_id          = azurerm_linux_virtual_machine.secondary.id
    endpoint_location    = "West US"
    priority             = 2
    provider             = azurerm.secondary
  }
}
```

---

3. GCP

Prerequisites

- GCP CLI configured with proper credentials.
- Terraform installed.
- Two GCP regions (Primary and Secondary).

Terraform Configuration

1. **Define Providers**

```
provider "google" {
  project = "your-project-id"
  region  = "us-central1"  # Primary region
}
provider "google" {
  alias   = "secondary"
  project = "your-project-id"
  region  = "us-west1"  # Secondary region
}
```

2. **Define Resources (e.g. Compute Instances Cloud Storage)**

```
resource "google_compute_instance" "primary" {
  name         = "primary-instance"
  zone         = "us-central1-a"
  machine_type = "e2-medium"
}
resource "google_compute_instance" "secondary" {
  name         = "secondary-instance"
  zone         = "us-west1-b"
  machine_type = "e2-medium"
  provider     = google.secondary
}
```

3. **Load Balancer for Automatic Failover**

```
resource "google_compute_global_forwarding_rule" "default" {
```

```
  name        = "default-http"
  target      = google_compute_target_http_proxy.default.id
  port_range  = "80"
}
resource "google_compute_target_http_proxy" "default" {
  url_map = google_compute_url_map.default.id
}
resource "google_compute_url_map" "default" {
  default_url_redirect {
    https_redirect = true
  }

  host_rule {
    hosts = ["example.com"]
    path_matcher = google_compute_path_matcher.default.id
  }
}

resource "google_compute_path_matcher" "default" {
  default_backend_service = google_compute_backend_service.primary.id
  path_rule {
    paths   = ["/*"]
    backend_service = google_compute_backend_service.primary.id
```

```
  }
}
```

---

4. Oracle Cloud Infrastructure (OCI)

Prerequisites

- OCI CLI configured with proper credentials.
- Terraform installed.
- Two OCI regions (Primary and Secondary).

Terraform Configuration

1. **Define Providers**

```
provider "oci" {
  region = "us-phoenix-1"  # Primary region
}
provider "oci" {
  alias  = "secondary"
  region = "us-ashburn-1"  # Secondary region
}
```

2. **Define Resources (e.g. Compute Instances Block Volumes)**

```
resource "oci_core_instance" "primary" {
  availability_domain = "phx-ad-1"
  compartment_id      = "your_compartment_id"
  shape               = "VM.Standard2.1"
}
```

```
resource "oci_core_instance" "secondary" {
  availability_domain = "ashburn-ad-1"
  compartment_id      = "your_compartment_id"
  shape               = "VM.Standard2.1"
  provider            = oci.secondary
}
```

3. **Failover Mechanism using OCI Load Balancer**

```
resource "oci_load_balancer" "primary" {
  compartment_id = "your_compartment_id"
  shape          = "100Mbps"
}
resource "oci_load_balancer" "secondary" {
  compartment_id = "your_compartment_id"
  shape          = "100Mbps"
  provider       = oci.secondary
}
```

Conclusion

By using Terraform you can automate the disaster recovery process across AWS Azure GCP and Oracle Cloud platforms. This setup includes provisioning resources setting up load balancers and defining failover rules to ensure high availability during disasters. Once this infrastructure is in place your cloud applications will be resilient ensuring minimal downtime.

---

3. Serverless

Project 1. Serverless Application Deployment

A serverless application allows you to run code without managing the underlying infrastructure. By using Terraform we can deploy serverless applications on various cloud providers including AWS Azure Google Cloud Platform (GCP) and Oracle Cloud. Serverless applications are typically event-driven meaning they are triggered by events like HTTP requests file uploads or database updates. This approach reduces infrastructure management overhead and enables scalable and cost-effective applications.

---

1. AWS Serverless Application Deployment

Prerequisites

- AWS account
- Terraform installed
- AWS CLI configured

Step-by-Step Configuration

Create a main.tf file

Define the provider and the serverless function.

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_lambda_function" "example" {
  function_name = "serverless-example"

  s3_bucket = "my-bucket"
  s3_key    = "lambda-code.zip"
```

```
  handler   = "index.handler"
  runtime   = "nodejs14.x"
}
resource "aws_api_gateway_rest_api" "example" {
  name        = "example-api"
  description = "Serverless API Gateway"
}
resource "aws_api_gateway_resource" "example" {
  rest_api_id = aws_api_gateway_rest_api.example.id
  parent_id   = aws_api_gateway_rest_api.example.root_resource_id
  path_part   = "example"
}
resource "aws_api_gateway_method" "example" {
  rest_api_id   = aws_api_gateway_rest_api.example.id
  resource_id   = aws_api_gateway_resource.example.id
  http_method   = "GET"
  authorization = "NONE"
}
resource "aws_api_gateway_integration" "example" {
  rest_api_id = aws_api_gateway_rest_api.example.id
  resource_id = aws_api_gateway_resource.example.id
  http_method = aws_api_gateway_method.example.http_method
  integration_http_method = "POST"
```

```
  type = "AWS_PROXY"
  uri = aws_lambda_function.example.invoke_arn
}
```

Initialize Terraform

```
terraform init
```

Apply the Configuration

```
terraform apply
```

Test the Serverless Application

After deployment you will get an endpoint URL for the API Gateway to test the Lambda function.

---

2. Azure Serverless Application Deployment

Prerequisites

- Azure account
- Terraform installed
- Azure CLI configured

Step-by-Step Configuration

Create a main.tf file

Define the provider and serverless function for Azure.

```
provider "azurerm" {
```

```
  features {}
}
resource "azurerm_function_app" "example" {
  name                       = "serverless-example-function"
  location                   = "East US"
  resource_group_name        = "example-resources"
  app_service_plan_id        = azurerm_app_service_plan.example.id
  storage_connection_string  = azurerm_storage_account.example.primary_connection_string
  version                    = "~3"

  site_config {
    linux_fx_version = "NODE|14-lts"
  }
}
resource "azurerm_storage_account" "example" {
  name                     = "examplestorageacct"
  resource_group_name      = "example-resources"
  location                 = "East US"
  account_tier              = "Standard"
  account_replication_type = "LRS"
}
resource "azurerm_app_service_plan" "example" {
  name                = "example-service-plan"
```

```
  location            = "East US"
  resource_group_name = "example-resources"
  kind                = "FunctionApp"
  reserved            = true
  sku {
    tier = "Dynamic"
    size = "Y1"
  }
}
```

Initialize Terraform
terraform init

Apply the Configuration
terraform apply

Test the Serverless Application

After deployment you'll have a URL for the Azure Function to test.

---

3. GCP Serverless Application Deployment

Prerequisites

- GCP account
- Terraform installed
- Google Cloud SDK configured

Step-by-Step Configuration

Create a main.tf file

Define the provider and the serverless function for GCP.

```
provider "google" {
  project = "your-project-id"
  region  = "us-central1"
}
resource "google_cloudfunctions_function" "example" {
  name        = "serverless-example"
  description = "GCP serverless function"
  runtime     = "nodejs14"
  entry_point = "helloWorld"
  available_memory_mb = 256
  source_archive_bucket = "your-bucket"
  source_archive_object = "function-code.zip"
}

resource "google_storage_bucket" "example" {
  name     = "your-bucket"
  location = "US"
}
```

Initialize Terraform
terraform init

Apply the Configuration
terraform apply

Test the Serverless Application

After deployment the function will have an HTTP endpoint to invoke it.

---

4. Oracle Cloud Serverless Application Deployment

Prerequisites

- Oracle Cloud account
- Terraform installed
- Oracle Cloud CLI configured

Step-by-Step Configuration

Create a main.tf file Define the provider and the serverless function for Oracle Cloud.

```
provider "oci" {
  region = "us-phoenix-1"
}
resource "oci_functions_function" "example" {
  application_id = oci_functions_application.example.id
  display_name   = "serverless-example"
  image          = "your-docker-image"
  memory_in_mbs  = 256
  timeout_in_seconds = 30
}
resource "oci_functions_application" "example" {
  compartment_id = "your-compartment-id"
  display_name   = "example-app"
  region         = "us-phoenix-1"
```

}

Initialize Terraform
terraform init

Apply the Configuration
terraform apply

Test the Serverless Application

After deployment the Oracle function will be accessible through an endpoint for testing.

Conclusion

With the above Terraform configurations we can deploy serverless applications on AWS Azure GCP and Oracle Cloud. These steps involve defining the cloud provider creating the necessary serverless functions configuring them with event triggers (like API Gateway or HTTP endpoint) and applying the configurations. This approach is scalable and each cloud provides specific tools to monitor and manage serverless applications effectively.

Project 2. Serverless Infrastructure with

AWS Lambda

Use Terraform to set up a serverless architecture with AWS Lambda API Gateway DynamoDB and other serverless components.

Creating serverless infrastructure with AWS Lambda and other components like API Gateway and DynamoDB is a popular use case for implementing event-driven architectures.Serverless computing enables developers to focus on writing code without worrying about managing servers. In this project you will deploy a serverless architecture using Terraform to automate the provisioning of resources. Each cloud provider's implementation involves their specific serverless services

- **AWS** Lambda API Gateway DynamoDB
- **Azure** Functions API Management Cosmos DB
- **GCP** Cloud Functions API Gateway Firestore
- **Oracle** Functions API Gateway Autonomous NoSQL Database

Terraform an Infrastructure as Code (IaC) tool will ensure consistent deployment across providers while managing dependencies and configurations.

Step-by-Step for AWS

Prerequisites

- AWS account
- Terraform CLI installed
- IAM user with admin access
- AWS CLI configured

Terraform Configuration

Initialize Provider
Configure the AWS provider with your region

```
provider "aws" {

  region = "us-east-1"

}
```

Create Lambda Function
Define a simple Lambda function

```
resource "aws_lambda_function" "example" {

  function_name = "example-function"

  runtime       = "python3.9"

  handler       = "lambda_function.lambda_handler"

  role          = aws_iam_role.lambda_exec.arn

  filename      = "function.zip"

}
```

```
resource "aws_iam_role" "lambda_exec" {
  name = "lambda_exec_role"
  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Action = "stsAssumeRole"
      Effect = "Allow"
      Principal = { Service = "lambda.amazonaws.com" }
    }]
  })
}
```

API Gateway
Create an API Gateway endpoint for the Lambda function

```
resource "aws_apigatewayv2_api" "api" {
  name          = "example-api"
  protocol_type = "HTTP"
}
```

DynamoDB Table
Provision a DynamoDB table

```
resource "aws_dynamodb_table" "example" {
  name         = "example-table"
  _key     = "id"
  attribute {
    name = "id"
```

```
  type = "S"
 }
 billing_mode = "PAY_PER_REQUEST"
}
```

Deployment Steps

- Save the files in a .tf file (e.g. main.tf).
- Run terraform init.
- Run terraform plan.
- Run terraform apply.

Step-by-Step for Azure

Terraform Configuration

Initialize Provider

```
provider "azurerm" {
 features {}
}
```

Create Azure Function

Use an App Service Plan for Functions

```
resource "azurerm_function_app" "example" {
 name                = "example-function"
 resource_group_name = azurerm_resource_group.example.name
 location            = azurerm_resource_group.example.location
 app_service_plan_id = azurerm_app_service_plan.example.id
}
```

API Management
Define API Management for routing requests

```
resource "azurerm_api_management" "example" {
  name                = "example-apim"
  resource_group_name = azurerm_resource_group.example.name
  location            = azurerm_resource_group.example.location
  publisher_name      = "example"
  publisher_email     = "example@example.com"
}
```

Cosmos DB
Provision a NoSQL Cosmos DB instance

```
resource "azurerm_cosmosdb_account" "example" {
  name                = "example-cosmos"
  resource_group_name = azurerm_resource_group.example.name
  location            = azurerm_resource_group.example.location
  offer_type          = "Standard"
  kind                = "GlobalDocumentDB"
}
```

GCP

Terraform Configuration

Initialize Provider

```
provider "google" {
  project = "your-project-id"
```

```
  region  = "us-central1"
}
```

Create Cloud Function

```
resource "google_cloudfunctions_function" "example" {
  name        = "example-function"
  runtime     = "nodejs14"
  entry_point = "helloWorld"
  source_archive_bucket = google_storage_bucket.bucket.name
  source_archive_object = google_storage_bucket_object.archive.name
  trigger_http = true
}
```

API Gateway

```
resource "google_api_gateway_api" "example" {
  api_id = "example-api"
}
```

Firestore Database

```
resource "google_firestore_database" "example" {
  name   = "(default)"
  region = "us-central"
}
```

Oracle Cloud Infrastructure (OCI)

Initialize Provider

Terraform init

Terraform Configuration

```
provider "oci" {

  tenancy_ocid       = "ocid1.tenancy.oc1..example"

  user_ocid          = "ocid1.user.oc1..example"

  fingerprint        = "your-fingerprint"

  private_key_path   = "/path/to/your/private_key.pem"

  region             = "us-ashburn-1"

}
```

Create Function

```
resource "oci_functions_function" "example" {

  application_id = oci_functions_application.example.id

  display_name   = "example-function"

}
```

API Gateway

```
resource "oci_apigateway_gateway" "example" {

  display_name   = "example-gateway"

  compartment_id = var.compartment_id

}
```

NoSQL Database

```
resource "oci_nosql_table" "example" {
```

```
  compartment_id = var.compartment_id

  name           = "example-table"

}
```

Summary

The steps for each cloud provider involve

- Configuring the provider.
- Defining the serverless function.
- Setting up API Gateway for routing requests.
- Adding a database to store data.

Terraform abstracts the complexities of resource provisioning enabling a declarative and repeatable infrastructure deployment process. Each configuration can be saved in separate .tf files and reused or modified for other projects.

Project 2. Containerized Microservices Architecture
Provision infrastructure for containerized applications and microservices including Docker containers Kubernetes and microservices architecture using Terraform.

AWS

Prerequisites

- AWS Account
- Terraform installed on your machine
- AWS CLI configured with access and secret keys

Steps

1. Set Up Terraform Directory Structure

- Create a directory
  `mkdir aws-microservices && cd aws-microservices`
- Subdirectories

- modules/
- main.tf
- variables.tf
- outputs.tf

2. Write Terraform Configurations

`main.tf`

Define provider

```
provider "aws" {

  region = var.aws_region

}
```

- Create VPC subnets and security groups
- Use ECS (Elastic Container Service) for container orchestration
- Set up an ECS cluster and Fargate service to run containers
- Use an Application Load Balancer for routing traffic

`variables.tf`

Define variables for AWS region VPC CIDR ECS configurations etc.

```
variable "aws_region" {

  default = "us-west-1"

}
```

`outputs.tf`

Output ECS Cluster ARN Load Balancer DNS etc.

```
output "ecs_cluster_arn" {

  value = aws_ecs_cluster.main.arn

}
```

3. Deploy Using Terraform

- Initialize `terraform init`
- Plan `terraform plan`
- Apply `terraform apply`

Azure

Prerequisites

- Azure Account
- Terraform and Azure CLI installed
- Azure CLI authenticated

Steps

1. Set Up Terraform Directory

- Similar to AWS

2. Write Terraform Configurations

`main.tf`
Define provider

```
provider "azurerm" {

  features {}

}
```

    - Create Resource Group Virtual Network and Subnets
    - Use AKS (Azure Kubernetes Service) for managing Kubernetes clusters
    - Deploy microservices containers on AKS

`Variables.tf`
Define variables for resource group name location and AKS configurations

```
variable "location" {

  default = "East US"

}
```

`Outputs.tf`
Output AKS Cluster Name and Resource Group

```
output "aks_cluster_name" {

  value = azurerm_kubernetes_cluster.main.name

}
```

3. Deploy Using Terraform

- Initialize `terraform init`
- Plan `terraform plan`
- Apply `terraform apply`

GCP

Prerequisites

- Google Cloud Platform account
- Terraform and gcloud CLI installed
- gcloud CLI authenticated

Steps

1. Set Up Terraform Directory

- Similar to AWS

2. Write Terraform Configurations

`main.tf`
Define provider

```
provider "google" {

  project = var.project_id

  region = var.region

}
```

- Create VPC subnets and firewall rules
- Use GKE (Google Kubernetes Engine) to manage Kubernetes clusters
- Deploy microservices containers on GKE

`Variables.tf`
Include variables for project ID region and GKE configurations

```
variable "project_id" {}

variable "region" {

  default = "us-central1"

}
```

`Outputs.tf`
Output GKE cluster name and external IP

```
output "gke_cluster_name" {

  value = google_container_cluster.main.name

}
```

3. Deploy Using Terraform

- Initialize `terraform init`
- Plan `terraform plan`
- Apply `terraform apply`

Oracle Cloud

Prerequisites

- Oracle Cloud Infrastructure (OCI) account
- Terraform CLI installed
- OCI CLI configured

Steps

1. Set Up Terraform Directory

- Similar to AWS

2. Write Terraform Configurations

`main.tf`
Define provider

```
provider "oci" {
  tenancy_ocid     = var.tenancy_ocid
  user_ocid        = var.user_ocid
  fingerprint      = var.fingerprint
  private_key_path = var.private_key_path
  region           = var.region
}
```

- Create VCN subnets and security lists
- Use OKE (Oracle Kubernetes Engine) for Kubernetes clusters
- Deploy microservices containers on OKE

`Variables.tf`

Define variables for tenancy OCID user OCID region and OKE configurations

```
variable "region" {

  default = "us-ashburn-1"

}
```

`outputs.tf`

Output OKE cluster name and public endpoint

```
output "oke_cluster_name" {

  value = oci_containerengine_cluster.main.name

}
```

3. Deploy Using Terraform

- Initialize `terraform init`
- Plan `terraform plan`
- Apply `terraform apply`

Key Considerations

- Secure sensitive data (e.g. credentials) using environment variables or secret management tools like iCorp Vault.
- Implement monitoring and logging for deployed clusters using Prometheus Grafana or cloud-specific tools.
- Ensure proper IAM roles and policies for secure access.

Project 3. Serverless Architecture with AWS API Gateway & Lambda

Use Terraform to set up an API Gateway with Lambda functions DynamoDB and other resources for building serverless applications.

Introduction to Serverless Architecture with AWS API Gateway & Lambda

Serverless architecture allows developers to build and deploy applications without managing infrastructure. Using AWS API Gateway Lambda and DynamoDB we can create a highly scalable and cost-efficient application. Terraform automates the provisioning of these resources enabling Infrastructure as Code (IaC).

This guide provides configurations and step-by-step instructions for setting up similar serverless architectures on AWS Azure GCP and Oracle Cloud using Terraform.

---

1. AWS

Terraform Configuration

```
provider "aws" {

  region = "us-east-1"

}

resource "aws_dynamodb_table" "users" {

  name           = "users-table"

  billing_mode   = "PAY_PER_REQUEST"

  _key       = "id"


  attribute {

    name = "id"

    type = "S"

  }

}
```

```
resource "aws_iam_role" "lambda_exec_role" {
  name = "lambda_exec_role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = "stsAssumeRole"
        Effect = "Allow"
        Principal = {
          Service = "lambda.amazonaws.com"
        }
      }
    ]
  })
}

resource "aws_iam_policy_attachment" "lambda_exec_policy" {
  name       = "lambda_exec_policy"
  roles      = [aws_iam_role.lambda_exec_role.name]
  policy_arn = "arnawsiamawspolicy/service-role/AWSLambdaBasicExecutionRole"
}
```

```
resource "aws_lambda_function" "api_handler" {
  function_name = "user-api-handler"
  role          = aws_iam_role.lambda_exec_role.arn
  handler       = "index.handler"
  runtime       = "nodejs14.x"

  filename         = "lambda.zip"
  source_code_ = filebase64sha256("lambda.zip")
}

resource "aws_api_gateway_rest_api" "api" {
  name = "user-api"
}

resource "aws_api_gateway_resource" "users" {
  rest_api_id = aws_api_gateway_rest_api.api.id
  parent_id   = aws_api_gateway_rest_api.api.root_resource_id
  path_part   = "users"
}

resource "aws_api_gateway_method" "users_method" {
  rest_api_id   = aws_api_gateway_rest_api.api.id
  resource_id   = aws_api_gateway_resource.users.id
```

```
  http_method   = "POST"
  authorization = "NONE"
}

resource "aws_api_gateway_integration" "users_integration" {
  rest_api_id             = aws_api_gateway_rest_api.api.id
  resource_id             = aws_api_gateway_resource.users.id
  http_method             = aws_api_gateway_method.users_method.http_method
  integration_http_method = "POST"
  type                    = "AWS_PROXY"
  uri                     = aws_lambda_function.api_handler.invoke_arn
}

resource "aws_lambda_permission" "api_gateway" {
  statement_id  = "AllowAPIGatewayInvoke"
  action        = "lambdaInvokeFunction"
  function_name = aws_lambda_function.api_handler.arn
  principal     = "apigateway.amazonaws.com"
}
```

Step-by-Step Guide

- Set up AWS credentials and install Terraform.
- Write and apply the configuration to create resources.
- Package your Lambda function (e.g. zip -r lambda.zip index.js).
- Use terraform apply to deploy resources.

- Test the API Gateway endpoint.

---

2. Azure

Terraform Configuration

```
provider "azurerm" {
  features = {}
}

resource "azurerm_resource_group" "example" {
  name     = "example-resources"
  location = "East US"
}

resource "azurerm_storage_account" "example" {
  name                     = "examplestorageacct"
  resource_group_name      = azurerm_resource_group.example.name
  location                 = azurerm_resource_group.example.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}

resource "azurerm_app_service_plan" "example" {
```

```
  name                = "example-plan"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  kind                = "FunctionApp"
  sku {
    tier = "Dynamic"
    size = "Y1"
  }
}

resource "azurerm_function_app" "example" {
  name                       = "example-funcapp"
  location                   = azurerm_resource_group.example.location
  resource_group_name        = azurerm_resource_group.example.name
  app_service_plan_id        = azurerm_app_service_plan.example.id
  storage_account_name       = azurerm_storage_account.example.name
  storage_account_access_key = azurerm_storage_account.example.primary_access_key
}
```

Step-by-Step Guide

- Install Azure CLI and authenticate.
- Configure Azure Storage and Function App using Terraform.
- Deploy the configuration using terraform apply.
- Use Azure Portal or CLI to test your Function App.

---

3. GCP

Terraform Configuration

```
provider "google" {
  project = "your-project-id"
  region  = "us-central1"
}

resource "google_storage_bucket" "bucket" {
  name          = "example-function-bucket"
  location      = "US"
  force_destroy = true
}

resource "google_cloudfunctions_function" "function" {
  name        = "example-function"
  runtime     = "nodejs14"
  entry_point = "helloWorld"
  source_archive_bucket = google_storage_bucket.bucket.name
  source_archive_object = "function-source.zip"
  trigger_http = true
}
```

```
resource "google_cloud_run_service" "api" {
  name     = "example-api"
  location = "us-central1"

  template {
    spec {
      containers {
        image = "gcr.io/cloud-run/hello"
      }
    }
  }
}
```

Step-by-Step Guide

- Set up Google Cloud SDK and authenticate.
- Use Terraform to configure Cloud Functions and Cloud Run.
- Upload the function archive to Cloud Storage.
- Deploy the configuration and test the HTTP trigger.

---

4. Oracle Cloud

Terraform Configuration

```
provider "oci" {}
```

```
resource "oci_objectstorage_bucket" "bucket" {
  name          = "function-code-bucket"
  compartment_id = var.compartment_id
}

resource "oci_functions_application" "application" {
  display_name  = "ExampleApplication"
  compartment_id = var.compartment_id
}

resource "oci_functions_function" "function" {
  display_name   = "ExampleFunction"
  application_id = oci_functions_application.application.id
  memory_in_mbs  = 128
  timeout_in_seconds = 30

  image = "iad.ocir.io/your-region/function-image:latest"
}
```

Step-by-Step Guide

- Authenticate with Oracle Cloud CLI.
- Configure Object Storage for storing function code.
- Deploy a containerized function to Oracle Functions.
- Test the function using Oracle Console or CLI.

Summary

These Terraform configurations and steps show how to implement a serverless architecture across AWS Azure GCP and Oracle Cloud. While the exact resources differ the process demonstrates how IaC simplifies multi-cloud deployments.

Project 4. Serverless Event-Driven Architecture

Implement an event-driven architecture using AWS Lambda SNS SQS and DynamoDB with Terraform for managing serverless resources.

Creating a **Serverless Event-Driven Architecture** is an excellent use case for modern cloud environments. Below I'll outline the steps to implement this project on **AWS Azure GCP** and **Oracle Cloud** using **Terraform** along with small introductions for each platform.

Introduction to Event-Driven Architecture

An **event-driven architecture** relies on events (state changes or updates) to trigger and communicate between services. This architecture ensures scalability, decoupling of services and efficient resource utilization. For this project

- **AWS** Use Lambda for compute SNS for event publishing SQS for queueing and DynamoDB for data storage.
- **Azure** Use Azure Functions Event Grid for event routing Storage Queue for message queueing and Cosmos DB.
- **GCP** Use Cloud Functions Pub/Sub for messaging Cloud Tasks for queuing and Firestore.
- **Oracle Cloud** Use Oracle Functions Notifications Queue and Autonomous JSON Database.

---

1. AWS Implementation with Terraform

Steps

1. **Set Up Terraform Provider**
   Configure the AWS provider in your Terraform script.
2. **Create SNS Topic**
   Define an SNS topic to publish events.
3. **Create SQS Queue**
   Define an SQS queue to process the messages.
4. **Create Lambda Function**
   Write a Lambda function triggered by the SQS queue.
5. **Create DynamoDB Table**
   Define a DynamoDB table to store processed data.

Terraform Configuration

```
provider "aws" {

  region = "us-east-1"

}


resource "aws_sns_topic" "events_topic" {

  name = "event-driven-topic"

}


resource "aws_sqs_queue" "events_queue" {

  name = "event-driven-queue"

}


resource "aws_lambda_function" "process_event" {

  function_name = "process_event_function"

  runtime       = "nodejs18.x"
```

```
  role          = aws_iam_role.lambda_role.arn
  handler       = "index.handler"
  filename      = "lambda_function.zip"
}

resource "aws_dynamodb_table" "event_data" {
  name          = "event_data"
  _key       = "id"
  attribute {
    name = "id"
    type = "S"
  }
  billing_mode = "PAY_PER_REQUEST"
}
```

Execution Steps

- Initialize Terraform with terraform init.
- Apply configurations using terraform apply.
- Test the architecture by publishing events to SNS.

---

2. Azure Implementation with Terraform

Steps

1. **Set Up Azure Provider**
   Use Terraform to configure Azure credentials and initialize the provider.

2. **Create Event Grid**
   Set up an Event Grid topic to handle event publishing.
3. **Create Storage Queue**
   Configure a Storage Queue for message processing.
4. **Deploy Azure Function**
   Create an Azure Function triggered by the Storage Queue.
5. **Create Cosmos DB**
   Deploy Cosmos DB to store the processed data.

Terraform Configuration

```
provider "azurerm" {

  features {}

}


resource "azurerm_eventgrid_topic" "events_topic" {

  name                = "event-driven-topic"

  resource_group_name = azurerm_resource_group.rg.name

  location            = azurerm_resource_group.rg.location

}


resource "azurerm_storage_queue" "events_queue" {

  name                 = "eventqueue"

  storage_account_name  = azurerm_storage_account.storage.name

}


resource "azurerm_function_app" "process_event" {

  name                     = "process-event-function"
```

```
  resource_group_name       = azurerm_resource_group.rg.name
  location                  = azurerm_resource_group.rg.location
  storage_account_name      = azurerm_storage_account.storage.name
  storage_account_access_key = azurerm_storage_account.storage.primary_access_key
  app_service_plan_id       = azurerm_app_service_plan.service_plan.id
}

resource "azurerm_cosmosdb_account" "db" {
  name                = "cosmosdbaccount"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  offer_type          = "Standard"
  kind                = "GlobalDocumentDB"
}
```

---

3. GCP Implementation with Terraform

Steps

1. **Configure GCP Provider**
   Initialize the provider with your GCP credentials.
2. **Create Pub/Sub Topic**
   Define a Pub/Sub topic for event publishing.
3. **Create Cloud Function**
   Deploy a Cloud Function triggered by Pub/Sub.
4. **Set Up Firestore**
   Create a Firestore database to store processed events.

Terraform Configuration

```
provider "google" {
  project = "your-gcp-project-id"
  region  = "us-central1"
}

resource "google_pubsub_topic" "events_topic" {
  name = "event-driven-topic"
}

resource "google_cloudfunctions_function" "process_event" {
  name        = "process_event_function"
  description = "Processes events from Pub/Sub"
  runtime     = "nodejs16"
  entry_point = "processEvent"
  trigger_http = true

  source_archive_bucket = google_storage_bucket.code_bucket.name
  source_archive_object = "lambda_function.zip"
}

resource "google_firestore_document" "event_data" {
```

```
  name       = "event_document"
  collection = "events"
  fields = jsonencode({
    id   = "12345"
    data = "event_data"
  })
}
```

---

4. Oracle Cloud Implementation with Terraform

Steps

1. **Set Up OCI Provider**
   Use Terraform to configure OCI credentials.
2. **Create Notification Topic**
   Define a Notification topic for event publishing.
3. **Create Queue**
   Configure a Queue for event processing.
4. **Deploy Oracle Function**
   Set up an Oracle Function triggered by the queue.
5. **Create Autonomous JSON Database**
   Deploy an Autonomous JSON Database for storage.

Terraform Configuration

```
provider "oci" {
  tenancy_ocid     = "ocid1.tenancy.oc1..xxxx"
  user_ocid        = "ocid1.user.oc1..xxxx"
  fingerprint      = "your_fingerprint"
```

```
  private_key_path = "path_to_private_key"
  region           = "us-ashburn-1"
}

resource "oci_notification_topic" "events_topic" {
  name = "event-driven-topic"
}

resource "oci_queue" "events_queue" {
  name = "event-queue"
}

resource "oci_functions_function" "process_event" {
  display_name = "process_event_function"
  application_id = oci_functions_application.app.id
  image          = "function_image"
  memory_in_mbs  = 128
  timeout_in_seconds = 30
}

resource "oci_autonomous_database" "json_db" {
  db_name          = "JSONDB"
  admin_password   = "your_password"
```

```
  db_workload      = "OLTP"
  is_autonomous    = true
  data_storage_size_in_tbs = 1
}
```

Project 5. Event-Driven Serverless Microservices

Create a serverless microservices architecture using Terraform for AWS Lambda EventBridge and SQS integrating these components into an event-driven architecture.

creating **Event-Driven Serverless Microservices** using **Terraform** for AWS Azure GCP and Oracle. The architecture focuses on an **event-driven design** where components interact through events rather than direct calls ensuring flexibility and scalability.

Event-driven architectures decouple services by using events as triggers for actions. This project demonstrates how to implement an event-driven system using Terraform. We'll deploy serverless microservices that integrate the following components

- **Compute** Serverless functions (AWS Lambda Azure Functions Google Cloud Functions Oracle Functions).
- **Event Bus** Handles the routing of events (e.g. AWS EventBridge Azure Event Grid GCP Pub/Sub Oracle Events Service).
- **Queueing System** For asynchronous processing (e.g. SQS Azure Queue Storage GCP Pub/Sub Queues Oracle Streams).

The flow

1. An event is generated and published to the event bus.
2. The event bus routes the event to the appropriate serverless function.
3. The serverless function processes the event and optionally pushes messages to the queue for further processing.

---

1. Implementation for AWS

Components

- AWS Lambda
- EventBridge
- Amazon SQS

Step-by-Step

Step 1 Setup Terraform Configuration

Provider Configuration

```
provider "aws" {
  region = "us-east-1"
}
```

EventBridge Rule

```
resource "aws_cloudwatch_event_rule" "example_rule" {
  name        = "example-rule"
  event_pattern = <<EOF
{
  "source" ["my.service"]
  "detail-type" ["exampleEvent"]
}
EOF
}
```

Lambda Function

```
resource "aws_lambda_function" "example_lambda" {
```

```
  function_name = "exampleLambda"
  handler       = "lambda_function.lambda_handler"
  runtime       = "python3.9"

  filename         = "lambda.zip" # Package your function
  role             = aws_iam_role.lambda_exec.arn
}

resource "aws_iam_role" "lambda_exec" {
  name = "lambda_exec_role"
  assume_role_policy = <<EOF
{
  "Version" "2012-10-17"
  "Statement" [
    {
      "Action" "stsAssumeRole"
      "Effect" "Allow"
      "Principal" { "Service" "lambda.amazonaws.com" }
    }
  ]
}
EOF
}
```

SQS Queue

```
resource "aws_sqs_queue" "example_queue" {
  name = "example-queue"
}
```

Permissions and Event Targets

```
resource "aws_lambda_permission" "allow_eventbridge" {
  statement_id  = "AllowExecutionFromEventBridge"
  action        = "lambdaInvokeFunction"
  function_name = aws_lambda_function.example_lambda.function_name
  principal     = "events.amazonaws.com"
}

resource "aws_cloudwatch_event_target" "lambda_target" {
  rule      = aws_cloudwatch_event_rule.example_rule.name
  arn       = aws_lambda_function.example_lambda.arn
}
```

Step 2 Deploy and Test

- Deploy using terraform apply.
- Trigger an event in EventBridge using the AWS Console or CLI and verify processing.

---

2. Implementation for Azure

Components

- Azure Functions
- Event Grid
- Azure Queue Storage

Step-by-Step

Step 1 Setup Terraform Configuration

Provider Configuration

```
provider "azurerm" {

  features = {}

}
```

Storage Account and Queue

```
resource "azurerm_storage_account" "example" {

  name                     = "examplestorageacct"

  resource_group_name      = azurerm_resource_group.example.name

  location                 = "East US"

  account_tier             = "Standard"

  account_replication_type = "LRS"

}


resource "azurerm_storage_queue" "example" {

  name                 = "example-queue"

  storage_account_name = azurerm_storage_account.example.name
```

```
}
```

Azure Function

```
resource "azurerm_function_app" "example" {

  name                       = "example-func-app"

  location                   = azurerm_resource_group.example.location

  resource_group_name        = azurerm_resource_group.example.name

  storage_account_name       = azurerm_storage_account.example.name

  storage_account_access_key = azurerm_storage_account.example.primary_access_key

  app_service_plan_id        = azurerm_app_service_plan.example.id

}
```

Event Grid

```
resource "azurerm_eventgrid_system_topic" "example" {

  name                = "example-system-topic"

  location            = azurerm_resource_group.example.location

  resource_group_name = azurerm_resource_group.example.name

  source_arm_resource_id = azurerm_storage_account.example.id

}
```

Step 2 Deploy and Test

- Deploy with terraform apply.
- Publish an event to Event Grid and verify function execution.

---

3. Implementation for GCP

Components

- Google Cloud Functions
- Pub/Sub

Step-by-Step

Step 1 Setup Terraform Configuration

Provider Configuration

```
provider "google" {

  project = "my-project-id"

  region  = "us-central1"

}
```

Pub/Sub Topic

```
resource "google_pubsub_topic" "example_topic" {

  name = "example-topic"

}
```

Cloud Function

```
resource "google_cloudfunctions_function" "example_function" {

  name        = "exampleFunction"

  runtime     = "python310"

  entry_point = "handler"


  source_archive_bucket = google_storage_bucket.example.name
```

```
  source_archive_object = google_storage_bucket_object.example.name
}
```

Step 2 Deploy and Test

- Deploy with terraform apply.
- Publish messages to Pub/Sub and observe function invocation.

---

4. Implementation for Oracle Cloud

Components

- Oracle Functions
- Oracle Events
- Oracle Streams

Step-by-Step

Step 1 Setup Terraform Configuration

Provider Configuration

```
provider "oci" {
  tenancy_ocid     = var.tenancy_ocid
  user_ocid        = var.user_ocid
  fingerprint      = var.fingerprint
  private_key_path = var.private_key_path
  region           = var.region
}
```

Oracle Stream

```
resource "oci_streaming_stream" "example_stream" {
```

```
  name           = "example-stream"
  compartment_id = var.compartment_id
  partitions     = 1
}
```

Oracle Function

```
resource "oci_functions_function" "example_function" {
  display_name = "exampleFunction"
  application_id = oci_functions_application.example.id
}
```

Event Trigger

```
resource "oci_events_rule" "example_rule" {
  display_name = "example-rule"
  compartment_id = var.compartment_id
  condition = <<EOF
{
  "eventType" "com.oraclecloud.objectstorage.createobject"
}
EOF
}
```

Step 2 Deploy and Test

- Deploy using terraform apply.
- Trigger an event and monitor the function execution.

Conclusion

This guide demonstrates the flexibility of Terraform for managing event-driven microservices across major cloud platforms.

---

4. Hybrid Cloud

Project 1. Hybrid Cloud Infrastructure

This project involves configuring a hybrid cloud infrastructure that integrates both on-premises and cloud resources using Terraform. The goal is to leverage multiple cloud providers like AWS Azure GCP and Oracle Cloud while maintaining resources on-premises all managed from a single Terraform configuration

Hybrid cloud infrastructure is a combination of on-premises data centers and public cloud services that work together. Using Terraform we can manage infrastructure across multiple cloud providers and on-premises environments in a consistent and automated way. By using Terraform's multi-provider capability we can integrate AWS Azure Google Cloud Oracle Cloud and on-premises infrastructure (e.g. VMware) into a unified deployment pipeline.

Prerequisites

1. **Terraform Installed** Make sure Terraform is installed on your machine.
2. **Cloud Accounts** Have access to AWS Azure GCP and Oracle Cloud accounts.
3. **API Keys and Access Credentials** Ensure you have the necessary credentials for each provider (AWS access key Azure service principal GCP service account Oracle Cloud credentials).

Step 1 Setup AWS Provider

To use AWS as a cloud provider in Terraform you need the AWS access credentials and the region where you wish to deploy resources.

Terraform Configuration for AWS

```
provider "aws" {
  access_key = "YOUR_AWS_ACCESS_KEY"
  secret_key = "YOUR_AWS_SECRET_KEY"
  region     = "us-west-2"
}
```

1. Create an AWS IAM User with programmatic access.
2. Set up your AWS credentials either by environment variables or the AWS credentials file.
3. Use Terraform to deploy an AWS resource like an EC2 instance.

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Step 2 Setup Azure Provider

For integrating Azure you'll need an Azure service principal that has access to your subscription.

Terraform Configuration for Azure

```
provider "azurerm" {
  client_id       = "YOUR_AZURE_CLIENT_ID"
  client_secret   = "YOUR_AZURE_CLIENT_SECRET"
  tenant_id       = "YOUR_AZURE_TENANT_ID"
  subscription_id = "YOUR_AZURE_SUBSCRIPTION_ID"
  features        = {}
}
```

1. Create a service principal in Azure.

2. Set environment variable**s** for authentication or provide the credentials in the configuration.
3. Use Terraform to deploy an Azure resource like a Virtual Machine.

```
resource "azurerm_virtual_machine" "example" {
  name                = "example-vm"
  location            = "East US"
  resource_group_name = "example-resources"
  network_interface_ids = [azurerm_network_interface.example.id]
  vm_size             = "Standard_B1ms"
}
```

Step 3 Setup GCP Provider

To connect to GCP you need a service account with necessary permissions for resource management.

Terraform Configuration for GCP

```
provider "google" {
  credentials = file("path_to_your_service_account_file.json")
  project     = "your-project-id"
  region      = "us-central1"
}
```

1. **Create a service account** in the Google Cloud Console.
2. **Download the JSON key** for the service account.
3. Deploy a GCP resource like a Compute Engine instance.

```
resource "google_compute_instance" "example" {
 name         = "example-instance"
 machine_type = "f1-micro"
 zone         = "us-central1-a"

 boot_disk {
  initialize_params {
   image = "debian-cloud/debian-9"
  }
 }

 network_interface {
  network = "default"
  access_config {
  }
 }
}
```

Step 4 Setup Oracle Cloud Provider

Oracle Cloud requires a configuration file and the proper credentials.

Terraform Configuration for Oracle Cloud

```
provider "oci" {
```

```
  tenancy_ocid       = "YOUR_TENANCY_OCID"
  user_ocid          = "YOUR_USER_OCID"
  fingerprint        = "YOUR_FINGERPRINT"
  private_key_path   = "path_to_your_private_key.pem"
  region             = "us-phoenix-1"
}
```

1. **Create a user and generate keys** for authentication in Oracle Cloud.
2. **Configure the credentials** using environment variables or in the configuration file.
3. Create a simple OCI resource like a compute instance.

```
resource "oci_core_instance" "example" {
  availability_domain = "UocmPHX-AD-1"
  compartment_id      = "ocid1.compartment.oc1..aaaaaaaaa"
  display_name        = "example-instance"
  shape               = "VM.Standard2.1"
  subnet_id           = oci_core_subnet.example.id

  create_vnic_details {
    assign_public_ip = true
    subnet_id        = oci_core_subnet.example.id
  }
}
```

Step 5 On-Premises Integration (Using VMware as an Example)

To integrate on-premises resources you can use the **VMware provider** in Terraform. You'll need to connect to your VMware vSphere instance.

Terraform Configuration for VMware

```
provider "vsphere" {
  user           = "your-vsphere-username"
  password       = "your-vsphere-password"
  vsphere_server = "your-vsphere-server"

  allow_unverified_ssl = true
}
```

1. **Set up vSphere credentials** for connecting to your on-prem VMware environment.
2. Define resources such as virtual machines in VMware.

```
resource "vsphere_virtual_machine" "vm" {
  name             = "example-vm"
  resource_pool_id = data.vsphere_resource_pool.pool.id
  datastore_id     = data.vsphere_datastore.datastore.id

  num_cpus = 2
  memory   = 2048
  guest_id = "otherGuest"

  disk {
    label            = "disk0"
    size             = 10
    eagerly_scrub    = false
```

```
    thin_provisioned = true
  }

  network_interface {
    network_id   = data.vsphere_network.network.id
    adapter_type = "vmxnet3"
  }

  clone {
    template_uuid = data.vsphere_virtual_machine.template.id

    customize {
      linux_options {
        host_name = "example-vm"
        domain    = "local"
      }
    }
  }
}
```

Conclusion

- This hybrid cloud setup enables you to manage resources across multiple cloud providers and on-premises systems from a single Terraform configuration.

- we can deploy and manage a wide range of infrastructure components across AWS Azure GCP Oracle Cloud and VMware environments.
- With Terraform we have a powerful tool for defining infrastructure as code making management more consistent and reproducible.

Project 2. Hybrid Cloud Setup

To integrate resources across these cloud providers we would typically use **data sources** to reference and manage resources that might span multiple clouds along with managing connections to on-prem infrastructure. For example we can define VPC peering VPNs or private connections between these clouds and on-premises systems.

1. AWS Configuration

Directory Structure

```
hybrid-cloud-aws/
├── main.tf
├── providers.tf
├── variables.tf
├── outputs.tf
```

Files

- main.tf

```
resource "aws_vpc" "main" {
  cidr_block = var.vpc_cidr
  tags = {
```

```
    Name = "Hybrid-AWS-VPC"
  }
}

resource "aws_instance" "example" {
  ami           = var.ami_id
  instance_type = var.instance_type
  subnet_id     = aws_vpc.main.default_subnet_id

  tags = {
    Name = "AWS-Instance"
  }
}
```

- **providers.tf**

```
provider "aws" {
  region = var.aws_region
}
```

- **variables.tf**

```
variable "aws_region" {}
variable "vpc_cidr" { default = "10.0.0.0/16" }
```

```
variable "ami_id" {}

variable "instance_type" { default = "t2.micro" }
```

- **outputs.tf**

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


output "instance_id" {
  value = aws_instance.example.id
}
```

---

2. Azure Configuration

Directory Structure

```
hybrid-cloud-azure/
├── main.tf
├── providers.tf
├── variables.tf
├── outputs.tf
```

Files

- main.tf

```
resource "azurerm_resource_group" "main" {
  name     = var.resource_group_name
  location = var.location
}

resource "azurerm_virtual_network" "main" {
  name                = "Hybrid-Azure-VNet"
  resource_group_name = azurerm_resource_group.main.name
  location            = azurerm_resource_group.main.location
  address_space       = ["10.1.0.0/16"]
}
```

- providers.tf

```
provider "azurerm" {
  features = {}
}
```

- variables.tf

```
variable "resource_group_name" { default = "Hybrid-Azure-RG" }
variable "location" { default = "East US" }
```

- outputs.tf

```
output "resource_group_name" {
  value = azurerm_resource_group.main.name
}

output "vnet_name" {
  value = azurerm_virtual_network.main.name
}
```

---

3. GCP Configuration

Directory Structure

css

```
hybrid-cloud-gcp/
├── main.tf
├── providers.tf
├── variables.tf
├── outputs.tf
```

Files

- main.tf

```
resource "google_compute_network" "main" {
  name                    = "Hybrid-GCP-Network"
  auto_create_subnetworks = false
}

resource "google_compute_instance" "example" {
  name         = "gcp-instance"
  machine_type = var.machine_type
  zone         = var.zone

  boot_disk {
    initialize_params {
      image = var.image
    }
  }

  network_interface {
    network = google_compute_network.main.name
  }
}
```

- providers.tf

```
provider "google" {
  project = var.project
  region  = var.region
}
```

- variables.tf

```
variable "project" {}
variable "region" {}
variable "zone" {}
variable "machine_type" { default = "e2-medium" }
variable "image" {}
```

- outputs.tf

```
output "network_name" {
  value = google_compute_network.main.name
}

output "instance_name" {
  value = google_compute_instance.example.name
}
```

---

4. Oracle Cloud Infrastructure (OCI) Configuration

Directory Structure

```
hybrid-cloud-oci/
├── main.tf
├── providers.tf
├── variables.tf
├── outputs.tf
```

Files

- main.tf

```
resource "oci_core_vcn" "main" {
  cidr_block = var.cidr_block
  display_name = "Hybrid-OCI-VCN"
  compartment_id = var.compartment_id
}
resource "oci_core_instance" "example" {
  availability_domain = var.availability_domain
  compartment_id      = var.compartment_id
  shape               = var.shape
```

```
  create_vnic_details {
    subnet_id = var.subnet_id
  }
  source_details {
    source_type = "image"
    source_id   = var.image_id
  }
}
```

- providers.tf

```
provider "oci" {
  tenancy     = var.tenancy_ocid
  user        = var.user_ocid
  fingerprint = var.fingerprint
  region      = var.region
  private_key = file(var.private_key_path)
}
```

- variables.tf

```
variable "compartment_id" {}
variable "cidr_block" { default = "10.3.0.0/16" }
variable "availability_domain" {}
variable "shape" { default = "VM.Standard.E2.1" }
```

```
variable "image_id" {}

variable "subnet_id" {}

variable "tenancy_ocid" {}

variable "user_ocid" {}

variable "fingerprint" {}

variable "region" {}

variable "private_key_path" {}
```

- outputs.tf

```
output "vcn_name" {
  value = oci_core_vcn.main.display_name
}

output "instance_name" {
  value = oci_core_instance.example.id
}
```

Usage

Navigate to each directory and initialize Terraform

```
terraform init
```

Apply the configuration

```
terraform apply
```

Project 3. Infrastructure as Code for Docker Swarm

Use Terraform to provision a Docker

Swarm cluster including the setup of nodes networks and services for containerized applications.

1. Folder Structure

Organize your project for clarity

```
docker-swarm-iac/
├── aws/
│   ├── main.tf
│   ├── variables.tf
│   └── outputs.tf
├── azure/
│   ├── main.tf
│   ├── variables.tf
│   └── outputs.tf
├── gcp/
│   ├── main.tf
│   ├── variables.tf
```

```
│   └── outputs.tf
└── oci/
    ├── main.tf
    ├── variables.tf
    └── outputs.tf
```

2. Terraform Configuration

Each provider requires a specific setup. Below are examples for setting up Docker Swarm in each cloud.

AWS (aws/main.tf)

```
provider "aws" {
  region = var.region
}

resource "aws_instance" "docker_swarm_manager" {
  ami           = "ami-0c02fb55956c7d316" # Update with your AMI ID
  instance_type = "t2.micro"
  count         = var.manager_count

  tags = {
    Name = "Docker-Swarm-Manager"
  }

  user_data = <<-EOF
```

```
            #!/bin/
            apt-get update
            apt-get install -y docker.io
            docker swarm init
            EOF
}
resource "aws_instance" "docker_swarm_worker" {
  ami           = "ami-0c02fb55956c7d316"
  instance_type = "t2.micro"
  count         = var.worker_count

  tags = {
    Name = "Docker-Swarm-Worker"
  }

  user_data = <<-EOF
            #!/bin/
            apt-get update
            apt-get install -y docker.io
            EOF
}
```

Azure (azure/main.tf)

```
provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "main" {
  name     = var.resource_group_name
  location = var.location
}

resource "azurerm_virtual_machine" "docker_swarm_manager" {
  count               = var.manager_count
  name                = "docker-swarm-manager-${count.index}"
  location            = azurerm_resource_group.main.location
  resource_group_name   = azurerm_resource_group.main.name
  network_interface_ids = [azurerm_network_interface.main.id]
  vm_size             = "Standard_B1s"

  storage_os_disk {
    name              = "osdisk-${count.index}"
    caching           = "ReadWrite"
    create_option     = "FromImage"
    managed_disk_type = "Standard_LRS"
```

```
  }

  os_profile {
    computer_name  = "docker-swarm-manager-${count.index}"
    admin_username = var.admin_username
    admin_password = var.admin_password
  }

  os_profile_linux_config {
    disable_password_authentication = false
  }

  source_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "18.04-LTS"
    version   = "latest"
  }
}
```

Google Cloud Platform (gcp/main.tf)

```
provider "google" {
  project = var.project_id
```

```
  region  = var.region
}

resource "google_compute_instance" "docker_swarm_manager" {
  count        = var.manager_count
  name         = "docker-swarm-manager-${count.index}"
  machine_type = "e2-micro"
  zone         = var.zone

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-11"
    }
  }

  network_interface {
    network = "default"
    access_config {
    }
  }
  metadata_startup_script = <<-EOT
    #!/bin/
    apt-get update
```

```
    apt-get install -y docker.io
    docker swarm init
  EOT
}
```

Oracle Cloud Infrastructure (oci/main.tf)

```
provider "oci" {
  tenancy_ocid     = var.tenancy_ocid
  user_ocid        = var.user_ocid
  fingerprint      = var.fingerprint
  private_key_path = var.private_key_path
  region           = var.region
}
resource "oci_core_instance" "docker_swarm_manager" {
  count               = var.manager_count
  availability_domain =
data.oci_identity_availability_domains.ADs.availability_domains[0].name
  compartment_id      = var.compartment_id
  shape               = "VM.Standard.E2.1.Micro"

  metadata = {
    ssh_authorized_keys = file("~/.ssh/id_rsa.pub")
  }
```

```
  source_details {
    source_type = "image"
    source_id   = data.oci_core_images.default_image.id
  }

  create_vnic_details {
    assign_public_ip = true
    subnet_id        = var.subnet_id
  }

  user_data = base64encode(<<-EOT
    #!/bin/
    apt-get update
    apt-get install -y docker.io
    docker swarm init
  EOT
  )
}
```

---

3. Variables (variables.tf)

Each cloud configuration should define variables specific to its provider e.g. region instance count and credentials.

Example

```
variable "manager_count" {
  default = 1
}

variable "worker_count" {
  default = 2
}

variable "region" {
  default = "us-west-1" # Adjust per cloud
}
```

---

4. Outputs (outputs.tf)

Export useful outputs like instance IPs.

```
output "manager_ips" {
  value = aws_instance.docker_swarm_manager.*.public_ip
}
```

5. Running Terraform

Initialize Terraform for each provider

```
terraform init
```

Validate the configuration
terraform validate

Apply the configuration
terraform apply

Project 4. Cross-Cloud Application Deployment

Use Terraform to set up and deploy an application that spans multiple clouds (e.g. AWS for compute GCP for storage Azure for networking).

In this project we will set up and deploy an application that spans multiple clouds (AWS GCP Azure and Oracle Cloud). We will use Terraform to manage the infrastructure as code and deploy resources across different cloud providers. The goal of this project is to demonstrate how to use Terraform to provision resources such as compute instances storage and networking components from various cloud providers and connect them to run a cross-cloud application.

This project will have different steps for each cloud provider (AWS GCP Azure and Oracle) focusing on compute resources storage and networking.

---

1. AWS Configuration (Compute)

Prerequisites

- AWS account
- Terraform installed
- AWS CLI configured

Steps

1. **Create main.tf file** to configure AWS provider

```
provider "aws" {

  region = "us-west-2"

}
```

```
resource "aws_instance" "web_server" {
  ami           = "ami-0c55b159cbfafe1f0"  # Example Amazon Linux AMI
  instance_type = "t2.micro"
  tags = {
    Name = "WebServer"
  }
}

output "aws_instance_ip" {
  value = aws_instance.web_server.public_ip
}
```

2. **Initialize and apply Terraform**
    - Run terraform init to initialize the provider.
    - Run terraform apply to deploy the instance in AWS.

---

2. GCP Configuration (Storage)

Prerequisites

- GCP account
- Terraform installed
- GCP CLI configured

Steps

1. **Create main.tf file** to configure GCP provider

```
provider "google" {
```

```
  project = "your-gcp-project-id"
  region  = "us-central1"
}

resource "google_storage_bucket" "bucket" {
  name          = "my-cross-cloud-bucket"
  location      = "US"
  force_destroy = true
}

output "gcp_bucket_url" {
  value = google_storage_bucket.bucket.url
}
```

2. **Initialize and apply Terraform**
    - Run terraform init to initialize the GCP provider.
    - Run terraform apply to create the storage bucket in GCP.

---

3. Azure Configuration (Networking)

Prerequisites

- Azure account
- Terraform installed
- Azure CLI configured

Steps

1. **Create main.tf file** to configure Azure provider

```
provider "azurerm" {
  features {}
}
resource "azurerm_virtual_network" "vnet" {
  name                = "my-vnet"
  address_space       = ["10.0.0.0/16"]
  location            = "East US"
  resource_group_name = "my-resource-group"
}
resource "azurerm_subnet" "subnet" {
  name                 = "my-subnet"
  resource_group_name  = "my-resource-group"
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefix       = "10.0.1.0/24"
}
output "azure_vnet_id" {
  value = azurerm_virtual_network.vnet.id
}
```

2. **Initialize and apply Terraform**
    - Run terraform init to initialize the Azure provider.
    - Run terraform apply to create the virtual network and subnet in Azure.

---

4. Oracle Cloud Configuration (Compute)

Prerequisites

- Oracle Cloud account
- Terraform installed
- Oracle Cloud CLI configured

Steps

1. **Create main.tf file** to configure Oracle Cloud provider

```
provider "oci" {
  region = "us-phoenix-1"
}
resource "oci_core_instance" "my_instance" {
  availability_domain = "UocmPHX-AD-1"
  compartment_id      = "your-compartment-id"
  shape               = "VM.Standard2.1"
  display_name        = "MyOracleInstance"
  image               = "ocid1.image.oc1.phx.xxxxxxx"

  create_vnic_details {
    subnet_id = "your-subnet-id"
  }
}

output "oracle_instance_ip" {
  value = oci_core_instance.my_instance.public_ip
}
```

2. **Initialize and apply Terraform**
    - Run terraform init to initialize the Oracle Cloud provider.
    - Run terraform apply to deploy the compute instance in Oracle Cloud.

Project Structure Overview

You can create a folder structure for your Terraform configuration to manage different cloud configurations

css

```
project-root/
│
├── aws/
│   └── main.tf
│
├── gcp/
│   └── main.tf
│
├── azure/
│   └── main.tf
│
├── oracle/
│   └── main.tf
│
└── variables.tf
```

5. Connecting Cross-Cloud Resources

- **Networking** You can set up VPNs or VPC Peering between your cloud providers to allow communication between AWS GCP Azure and Oracle Cloud.
- **Application** The application deployed in AWS (compute) can access the storage in GCP or use networking from Azure.

Conclusion

This project demonstrates how to use Terraform to manage resources across multiple cloud providers and deploy a cross-cloud application. With this approach you can take advantage of specific features from each cloud provider such as AWS for compute GCP for storage Azure for networking and Oracle for compute. By using Terraform the entire infrastructure can be version-controlled and automated for consistency and scalability.

Project 5. Cross-Account Resource Sharing

Cross-Account Resource Sharing
Set up cross-account IAM roles for sharing S3 buckets or databases.

Project 6 Cross-Account Resource Sharing

Introduction Cross-Account Resource Sharing allows resources like S3 buckets databases and other services to be accessed by users or services in different AWS Azure GCP or Oracle accounts. This is useful in multi-account architectures where certain resources need to be shared between different accounts. In this project we'll set up cross-account access using Terraform configurations for AWS Azure GCP and Oracle.

---

1. AWS Cross-Account Resource Sharing (S3 Bucket Example)

Objective Grant access to an S3 bucket in one account to another AWS account.

Steps

1. **Create IAM Role in the Source Account (Account A)**
    - The source account will have the S3 bucket and we need to create an IAM role that allows the target account to access it.

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

resource "aws_iam_role" "cross_account_role" {
  name               = "cross-account-s3-access"
  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Effect    = "Allow"
        Principal = {
          AWS = "arnawsiamTARGET_ACCOUNT_IDroot"
        }
        Action    = "stsAssumeRole"
      }
    ]
  })
}
```

2. **Attach Policy to IAM Role**
    - Attach a policy that allows access to the S3 bucket.

```
resource "aws_iam_policy" "s3_access_policy" {
  name        = "s3-access-policy"
  description = "Policy for cross-account S3 access"
  policy      = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Effect   = "Allow"
        Action   = "s3GetObject"
        Resource = "arnawss3source-bucket/*"
      }
    ]
  })
}


resource "aws_iam_policy_attachment" "policy_attachment" {
  name       = "cross-account-s3-access-policy-attachment"
  policy_arn = aws_iam_policy.s3_access_policy.arn
  roles      = [aws_iam_role.cross_account_role.name]
}
```

3. **Assume Role in the Target Account (Account B)**
    - In the target account you'll need a mechanism (like an IAM policy) to assume the role and access the resources.

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

resource "aws_iam_role_policy" "cross_account_role_policy" {
  role   = "cross-account-iam-role"
  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Effect   = "Allow"
        Action   = "stsAssumeRole"
        Resource = "arnawsiamSOURCE_ACCOUNT_IDrole/cross-account-s3-access"
      }
    ]
  })
}
```

---

2. Azure Cross-Account Resource Sharing (Storage Blob Example)

Objective Allow cross-tenant access to an Azure Blob Storage container.

Steps

1. **Create a Service Principal in Source Tenant (Account A)**

```
provider "azurerm" {
  features {}
}

resource "azurerm_client_config" "example" {}

resource "azurerm_role_assignment" "example" {
  principal_id   = "TARGET_ACCOUNT_SERVICE_PRINCIPAL_ID"
  role_definition_name = "Storage Blob Data Contributor"
  scope = azurerm_storage_container.example.id
}
```

2. **Create the Storage Account and Container in Source Tenant (Account A)**

```
resource "azurerm_storage_account" "example" {
  name                     = "examplestorageacct"
  resource_group_name      = azurerm_resource_group.example.name
  location                 = "East US"
```

```
  account_tier             = "Standard"
  account_replication_type = "LRS"
}

resource "azurerm_storage_container" "example" {
  name                  = "example-container"
  storage_account_name  = azurerm_storage_account.example.name
  container_access_type = "private"
}
```

3. **Assume Role in Target Tenant (Account B)**

In the target tenant use the service principal to authenticate and access the storage.

```
resource "azurerm_role_assignment" "target" {
  principal_id   = "TARGET_ACCOUNT_SERVICE_PRINCIPAL_ID"
  role_definition_name = "Storage Blob Data Contributor"
  scope = azurerm_storage_container.example.id
}
```

---

3. GCP Cross-Account Resource Sharing (Storage Bucket Example)

Objective Grant access to a Google Cloud Storage bucket from another project (account).

Steps

1. **Create a Service Account in Source Project (Account A)**

```
provider "google" {
  credentials = file("<path-to-credentials-file>")
  project     = "source-project-id"
  region      = "us-central1"
}

resource "google_service_account" "example" {
  account_id   = "cross-account-access"
  display_name = "Cross Account Access Service Account"
}
```

2. **Grant IAM Permissions to the Service Account (Account A)**

```
resource "google_storage_bucket_iam_member" "member" {
  bucket = "source-bucket-name"
  role   = "roles/storage.objectViewer"
  member = "serviceAccount${google_service_account.example.email}"
}
```

3. **Grant the Service Account Access in the Target Account (Account B)**

```
resource "google_storage_bucket_iam_member" "target" {
  bucket = "target-bucket-name"
```

```
  role   = "roles/storage.objectViewer"
  member = "serviceAccount${google_service_account.example.email}"
}
```

---

4. Oracle Cross-Account Resource Sharing (Object Storage Example)

Objective Share an Oracle Cloud Object Storage bucket with another Oracle Cloud account.

Steps

1. **Create a Dynamic Group in the Source Oracle Cloud Account (Account A)**

```
provider "oci" {
  region = "us-phoenix-1"
}

resource "oci_identity_dynamic_group" "example" {
  compartment_id = "source-compartment-id"
  name           = "CrossAccountDynamicGroup"
  description    = "Dynamic group for cross-account resource sharing"
}
```

2. **Grant Permissions to the Dynamic Group**

```
resource "oci_identity_policy" "policy" {
  compartment_id = "source-compartment-id"
  name           = "CrossAccountPolicy"
  statements     = [
```

```
  "Allow dynamic-group CrossAccountDynamicGroup to read object-storage in compartment source-compartment-id"
 ]
}
```

3. **Create a Cross-Account User in the Target Account**

```
resource "oci_identity_user" "target_user" {
 compartment_id = "target-compartment-id"
 name           = "cross-account-user"
 description    = "User for cross-account access"
}
```

Conclusion

Each of the cloud providers (AWS Azure GCP and Oracle) has its own method for cross-account resource sharing. The above Terraform configurations allow you to set up roles permissions and service accounts to enable access to resources like S3 buckets storage containers or object storage across accounts. Always ensure that you follow the least privilege principle when assigning permissions.

---

5. Kubernetes & Containers

Project 1 Kubernetes Cluster with Terraform

This project will guide you in deploying a Kubernetes cluster on different cloud platforms (AWS Azure GCP and Oracle Cloud) using **Terraform**. You'll create the necessary resources like VPCs subnets and other infrastructure components for managing the Kubernetes cluster in each cloud environment.

Terraform Project Overview

Terraform is an open-source tool for infrastructure as code (IaC) that allows you to provision and manage cloud resources across various platforms. In this project you'll use Terraform to

- Set up a Kubernetes cluster on a cloud platform (AWS Azure GCP or Oracle).
- Automate the creation of related resources such as networking components (VPC subnets) and storage.
- Use terraform apply to provision your cluster and terraform destroy to tear it down.

Prerequisites

Before starting ensure you have the following

- **Terraform installed** Install Terraform.
- **Cloud provider account** Set up the respective account for AWS Azure GCP or Oracle Cloud.
- **Terraform Cloud provider configuration** Each cloud provider requires authentication so ensure you have the appropriate credentials (API keys service accounts etc.).

---

1. AWS - EKS (Elastic Kubernetes Service)

Step-by-Step for AWS EKS Setup with Terraform

1. **Set up your AWS credentials**
    - Use AWS CLI or set environment variables for your access and secret keys.

Example

export AWS_ACCESS_KEY_ID="your-access-key"

export AWS_SECRET_ACCESS_KEY="your-secret-key"

    -

Create the Terraform Configuration Create a new directory and inside it create a file named main.tf. Add the following content

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

resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "main" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.1.0/24"
  availability_zone = "us-west-2a"
}

resource "aws_eks_cluster" "eks_cluster" {
  name     = "my-eks-cluster"
  role_arn = "arnawsiam123456789012role/eksServiceRole"

  vpc_config {
    subnet_ids = [aws_subnet.main.id]
  }
```

```
}


resource "aws_eks_node_group" "node_group" {
  cluster_name    = aws_eks_cluster.eks_cluster.name
  node_group_name = "my-node-group"
  node_role_arn   = "arnawsiam123456789012role/eksNodeRole"
  subnet_ids      = [aws_subnet.main.id]

  scaling_config {
    desired_size = 2
    max_size     = 3
    min_size     = 1
  }
}
```

Initialize Terraform
terraform init

Apply the configuration
terraform apply

Connect to your Kubernetes cluster Use aws eks update-kubeconfig to update your kubeconfig file

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

Verify the connection
kubectl get nodes

---

2. Azure - AKS (Azure Kubernetes Service)

Step-by-Step for Azure AKS Setup with Terraform

1. **Set up your Azure credentials**
    - Use Azure CLI or set the environment variables for your subscription and client credentials.

Example

export AZURE_CLIENT_ID="your-client-id"

export AZURE_CLIENT_SECRET="your-client-secret"

export AZURE_SUBSCRIPTION_ID="your-subscription-id"

export AZURE_TENANT_ID="your-tenant-id"

Create the Terraform Configuration Create a new directory and inside it create a file named main.tf. Add the following content

```
provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "rg" {
  name     = "myResourceGroup"
  location = "East US"
```

```
}

resource "azurerm_virtual_network" "vnet" {
  name                = "myVnet"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  address_space       = ["10.0.0.0/16"]
}

resource "azurerm_subnet" "subnet" {
  name                 = "mySubnet"
  resource_group_name  = azurerm_resource_group.rg.name
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefixes     = ["10.0.1.0/24"]
}

resource "azurerm_kubernetes_cluster" "aks_cluster" {
  name                = "myakscluster"
  location            = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name
  dns_prefix          = "myakscluster"

  default_node_pool {
```

```
    name       = "default"
    node_count = 2
    vm_size    = "Standard_DS2_v2"
    vnet_subnet_id = azurerm_subnet.subnet.id
  }

  identity {
    type = "SystemAssigned"
  }
}
```

Initialize Terraform
terraform init

Apply the configuration
terraform apply

Connect to your Kubernetes cluster Use Azure CLI to get your kubeconfig
az aks get-credentials --resource-group myResourceGroup --name myakscluster

Verify the connection
kubectl get nodes

---

3. GCP - GKE (Google Kubernetes Engine)

Step-by-Step for GCP GKE Setup with Terraform

1. **Set up your GCP credentials**
    - Set the environment variable for your Google Cloud project ID and service account credentials

```
export GOOGLE_CLOUD_PROJECT="your-project-id"

export GOOGLE_APPLICATION_CREDENTIALS="path-to-your-service-account-key.json"
```

Create the Terraform Configuration Create a new directory and inside it create a file named main.tf. Add the following content

```
provider "google" {
  project = "your-project-id"
  region  = "us-central1"
}

resource "google_container_cluster" "primary" {
  name               = "my-gke-cluster"
  location           = "us-central1-a"
  initial_node_count = 3

  node_config {
    machine_type = "e2-medium"
  }
}
```

Initialize Terraform
terraform init

Apply the configuration
terraform apply

Connect to your Kubernetes cluster
gcloud container clusters get-credentials my-gke-cluster --region us-central1 --project your-project-id

Verify the connection
kubectl get nodes

---

4. Oracle Cloud - OKE (Oracle Kubernetes Engine)

Step-by-Step for Oracle OKE Setup with Terraform

1. **Set up your Oracle Cloud credentials**
    - Set the environment variables for your Oracle Cloud credentials

export OCI_PROFILE="your-profile"

export OCI_CONFIG_FILE="~/.oci/config"

Create the Terraform Configuration Create a new directory and inside it create a file named main.tf. Add the following content

provider "oci" {

  tenancy_ocid   = "your-tenancy-ocid"

```
  user_ocid        = "your-user-ocid"
  fingerprint      = "your-fingerprint"
  private_key_path = "~/.oci/oci_api_key.pem"
  region           = "us-phoenix-1"
}

resource "oci_containerengine_cluster" "oke_cluster" {
  compartment_id = "your-compartment-id"
  name           = "my-oke-cluster"
  vcn_id         = "your-vcn-id"
}
```

Initialize Terraform
terraform init

Apply the configuration
terraform apply

Connect to your Kubernetes cluster Retrieve the kubeconfig from Oracle Cloud and configure kubectl

oci ce cluster create-kubeconfig --cluster-id my-oke-cluster-id --file $HOME/.kube/config --region us-phoenix-1

Verify the connection
kubectl get nodes

Conclusion

This project enables you to provision a Kubernetes cluster on multiple cloud platforms using Terraform. Follow the platform-specific instructions to configure your cluster and verify its deployment.

Project 2. Managed Kubernetes Infrastructure

Here's a step-by-step guide to set up **Managed Kubernetes Infrastructure** using **Terraform** for **AWS (EKS) Azure (AKS) GCP (GKE)** and **Oracle (OKE)**.

---

1. AWS EKS (Elastic Kubernetes Service)

Step 1 Set up Terraform Provider

```
provider "aws" {

  region = "us-east-1"

}
```

Step 2 Create VPC

```
module "vpc" {

  source  = "terraform-aws-modules/vpc/aws"

  version = "3.19.0"


  name                 = "eks-vpc"

  cidr                 = "10.0.0.0/16"
```

```
  azs                  = ["us-east-1a" "us-east-1b"]
  public_subnets       = ["10.0.1.0/24" "10.0.2.0/24"]
  private_subnets      = ["10.0.3.0/24" "10.0.4.0/24"]
  enable_nat_gateway   = true
}
```

Step 3 Create EKS Cluster

```
module "eks" {
  source          = "terraform-aws-modules/eks/aws"
  cluster_name    = "eks-cluster"
  cluster_version = "1.26"
  subnets         = module.vpc.private_subnets
  vpc_id          = module.vpc.vpc_id

  node_groups = {
    eks_nodes = {
      desired_capacity = 2
      max_capacity     = 3
      min_capacity     = 1

      instance_type = "t3.medium"
    }
```

```
  }
}
```

Step 4 Output Kubernetes Configuration

```
output "kubeconfig" {
  value = module.eks.kubeconfig
}
```

---

2. Azure AKS (Azure Kubernetes Service)

Step 1 Set up Terraform Provider

```
provider "azurerm" {
  features {}
}
```

Step 2 Create Resource Group

```
resource "azurerm_resource_group" "rg" {
  name     = "aks-resource-group"
  location = "East US"
}
```

Step 3 Create AKS Cluster

```
resource "azurerm_kubernetes_cluster" "aks" {
  name                = "aks-cluster"
  location            = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name
  dns_prefix          = "k8s"

  default_node_pool {
    name       = "default"
    node_count = 2
    vm_size    = "Standard_DS2_v2"
  }

  identity {
    type = "SystemAssigned"
  }
}
```

Step 4 Output Kubernetes Configuration

```
output "kube_config" {
  value = azurerm_kubernetes_cluster.aks.kube_config_raw
```

```
}
```

---

3. GCP GKE (Google Kubernetes Engine)

Step 1 Set up Terraform Provider

```
provider "google" {
  project = "my-gcp-project-id"
  region  = "us-central1"
}
```

Step 2 Create VPC

```
resource "google_compute_network" "vpc" {
  name                    = "gke-vpc"
  auto_create_subnetworks = false
}
resource "google_compute_subnetwork" "subnet" {
  name          = "gke-subnet"
  ip_cidr_range = "10.0.0.0/24"
  region        = "us-central1"
  network       = google_compute_network.vpc.self_link
```

```
}
```

Step 3 Create GKE Cluster

```
resource "google_container_cluster" "gke" {
  name     = "gke-cluster"
  location = "us-central1"
  network  = google_compute_network.vpc.self_link
  subnetwork = google_compute_subnetwork.subnet.self_link

  initial_node_count = 3

  node_config {
    machine_type = "e2-medium"
  }
}
```

Step 4 Output Kubernetes Configuration

```
output "kubeconfig" {
  value = google_container_cluster.gke.kubeconfig
}
```

---

4. Oracle OKE (Oracle Kubernetes Engine)

Step 1 Set up Terraform Provider

```
provider "oci" {
  tenancy_ocid     = "ocid1.tenancy.oc1..example"
  user_ocid        = "ocid1.user.oc1..example"
  private_key_path = "~/.oci/oci_api_key.pem"
  fingerprint      = "fingerprint"
  region           = "us-ashburn-1"
}
```

Step 2 Create VCN and Subnets

```
resource "oci_core_virtual_network" "vcn" {
  cidr_block     = "10.0.0.0/16"
  display_name   = "oke-vcn"
  compartment_id = var.compartment_id
}

resource "oci_core_subnet" "subnet" {
  cidr_block       = "10.0.1.0/24"
  display_name     = "oke-subnet"
  vcn_id           = oci_core_virtual_network.vcn.id
  compartment_id   = var.compartment_id
```

```
  prohibit_public_ip_on_vnic = false
}
```

Step 3 Create OKE Cluster

```
resource "oci_containerengine_cluster" "oke" {
  name               = "oke-cluster"
  compartment_id     = var.compartment_id
  vcn_id             = oci_core_virtual_network.vcn.id
  kubernetes_version = "v1.26.0"
}
```

Step 4 Output Cluster Details

```
output "cluster_kubeconfig" {
  value = oci_containerengine_cluster.oke.kube_config
}
```

Summary

For each cloud provider

- Configure Terraform provider
- Set up networking (VPC/VCN and subnets)
- Deploy managed Kubernetes service
- Output the Kubernetes configuration for cluster management

Project 3. Managed Kubernetes with Helm

Use Terraform to deploy a Kubernetes cluster and manage Kubernetes applications using Helm charts.

Project Overview

This project will guide you through deploying a managed Kubernetes cluster on AWS Azure GCP and Oracle Cloud. Terraform will be used for provisioning the cloud infrastructure and Helm will be used to manage applications on the Kubernetes cluster.

Prerequisites

- **Terraform** installed on your system.
- **Helm** installed.
- Access to a cloud provider account (AWS Azure GCP Oracle Cloud) and the necessary credentials.
- Basic knowledge of Kubernetes and Helm.

Step-by-Step Guide

1. AWS - Managed Kubernetes with Helm

Step 1 Set up Terraform for AWS

- Install the AWS CLI and configure it with your AWS credentials (aws configure).
- Install Terraform if not done already.

Step 2 Terraform Configuration for EKS (Elastic Kubernetes Service)

```
provider "aws" {

  region = "us-west-2"

}



resource "aws_eks_cluster" "eks_cluster" {

  name     = "my-cluster"

  role_arn = "arnawsiam123456789012role/eks-cluster-role"
```

```
  vpc_config {
    subnet_ids = ["subnet-abc123" "subnet-def456"]
  }
}

resource "aws_eks_node_group" "eks_node_group" {
  cluster_name = aws_eks_cluster.eks_cluster.name
  node_role_arn = "arnawsiam123456789012role/eks-node-group-role"
  subnet_ids = ["subnet-abc123" "subnet-def456"]
  instance_type = "t3.medium"
  desired_size  = 2
}
```

Step 3 Initialize Terraform

Run the following commands to initialize and apply your Terraform configuration

terraform init

terraform apply

Step 4 Deploy Helm on AWS EKS

- Once the cluster is created configure kubectl to connect to the EKS cluster.

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

- **Use Helm to deploy an application**

helm install my-release bitnami/nginx

---

2. Azure - Managed Kubernetes with Helm

Step 1 Set up Terraform for Azure

- Install Azure CLI and configure it with your credentials (az login).
- Install Terraform.

Step 2 Terraform Configuration for Azure AKS (Azure Kubernetes Service)

```
provider "azurerm" {
  features {}
}

resource "azurerm_kubernetes_cluster" "aks_cluster" {
  name                = "my-cluster"
  location            = "East US"
  resource_group_name = "my-resource-group"
  kubernetes_version  = "1.21.2"

  default_node_pool {
    name       = "default"
    node_count = 2
    vm_size    = "Standard_DS2_v2"
  }
```

```
}

resource "azurerm_kubernetes_cluster_node_pool" "additional_pool" {
  name                = "additional-pool"
  kubernetes_cluster_id = azurerm_kubernetes_cluster.aks_cluster.id
  node_count          = 2
  vm_size             = "Standard_DS2_v2"
}
```

Step 3 Initialize Terraform

terraform init

terraform apply

Step 4 Deploy Helm on Azure AKS

- Once the cluster is created configure kubectl

az aks get-credentials --resource-group my-resource-group --name my-cluster

- Deploy an application using Helm

helm install my-release stable/nginx

---

3. GCP - Managed Kubernetes with Helm

Step 1 Set up Terraform for GCP

- Install Google Cloud SDK and configure it (gcloud init).
- Install Terraform.

Step 2 Terraform Configuration for GKE (Google Kubernetes Engine)

```
provider "google" {
  credentials = file("<YOUR-CREDENTIALS-FILE>.json")
  project     = "my-project-id"
  region      = "us-central1"
}


resource "google_container_cluster" "gke_cluster" {
  name     = "my-cluster"
  location = "us-central1-a"

  initial_node_count = 3

  node_config {
    machine_type = "n1-standard-2"
  }
}
```

Step 3 Initialize Terraform

terraform init

terraform apply

Step 4 Deploy Helm on GKE

- Once the cluster is created configure kubectl

gcloud container clusters get-credentials my-cluster --zone us-central1-a --project my-project-id

- **Deploy an application using Helm**

helm install my-release stable/nginx

---

4. Oracle Cloud - Managed Kubernetes with Helm

Step 1 Set up Terraform for Oracle Cloud

- Install Oracle Cloud CLI and configure it (oci setup config).
- Install Terraform.

Step 2 Terraform Configuration for Oracle OKE (Oracle Kubernetes Engine)

```
provider "oci" {
  tenancy_ocid       = "ocid1.tenancy.oc1..example"
  user_ocid          = "ocid1.user.oc1..example"
  fingerprint        = "fingerprint"
  private_key_path   = "~/.oci/oci_api_key.pem"
  region             = "us-phoenix-1"
}

resource "oci_containerengine_cluster" "oke_cluster" {
  name           = "my-cluster"
```

```
  compartment_id  = "ocid1.compartment.oc1..example"
  vcn_id          = "ocid1.vcn.oc1..example"
}

resource "oci_containerengine_node_pool" "oke_node_pool" {
  cluster_id      = oci_containerengine_cluster.oke_cluster.id
  name            = "my-node-pool"
  compartment_id  = "ocid1.compartment.oc1..example"
  node_shape      = "VM.Standard2.1"
  node_count      = 2
}
```

Step 3 Initialize Terraform

```
terraform init
terraform apply
```

Step 4 Deploy Helm on Oracle OKE

- Once the cluster is created configure kubectl

```
oci ce cluster kubeconfig --cluster-id my-cluster --file kubeconfig.yaml
export KUBECONFIG=kubeconfig.yaml
```

- **Deploy an application using Helm**

```
helm install my-release stable/nginx
```

Conclusion

This Terraform project allows you to set up managed Kubernetes clusters in various cloud providers (AWS Azure GCP Oracle) and deploy applications using Helm. You can extend this project by adding more configurations like custom Helm charts or advanced cluster settings to suit your specific use case.

Project 4. Terraform for Kubernetes Multi-Cluster Management

Project Overview

This project involves provisioning and managing multiple Kubernetes clusters across different cloud providers (AWS Azure GCP Oracle Cloud) using Terraform. By integrating with tools like Rancher we can centralize the management of these clusters making it easier to handle multi-region or multi-cloud environments. Terraform's Infrastructure as Code (IaC) approach allows for reproducible declarative configurations ensuring scalability and consistency across clusters.

Step-by-Step Guide

1. AWS Setup (with EKS)

Pre-requisites

- AWS Account with sufficient permissions.
- AWS CLI configured.
- Terraform installed.
- kubectl and eksctl installed.

Steps

1. **Create IAM Role and Policies for EKS**
    - Terraform will create the necessary IAM roles for Kubernetes to interact with AWS resources.

```
resource "aws_iam_role" "eks_cluster_role" {
  name = "eks-cluster-role"
  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = "stsAssumeRole"
        Effect = "Allow"
        Principal = {
          Service = "eks.amazonaws.com"
        }
      }
    ]
  })
}
```

Create EKS Cluster

- Define the AWS EKS cluster resource in Terraform.

```
resource "aws_eks_cluster" "example" {
  name     = "multi-cluster"
  role_arn = aws_iam_role.eks_cluster_role.arn
```

```
  vpc_config {
    subnet_ids = aws_subnet.subnet_ids[*].id
  }
}
```

2. **Create Worker Nodes**
    - You will also need worker nodes (EC2 instances) for your Kubernetes cluster.

```
resource "aws_eks_node_group" "example" {
  cluster_name    = aws_eks_cluster.example.name
  node_group_name = "example-node-group"
  node_role_arn   = aws_iam_role.eks_node_group_role.arn
  subnet_ids      = aws_subnet.subnet_ids[*].id
}
```

3. **Configure Rancher for Multi-Cluster Management**
    - Install Rancher and connect your clusters for central management.

---

2. Azure Setup (with AKS)

Pre-requisites

- Azure account with sufficient permissions.
- Azure CLI configured.
- Terraform installed.
- kubectl installed.

Steps

1. **Create a Resource Group**
    - Define the resource group for your Kubernetes clusters.

```
resource "azurerm_resource_group" "example" {
  name     = "multi-cluster-rg"
  location = "East US"
}
```

2. **Create AKS Cluster**
    - Define the AKS cluster in Terraform.

```
resource "azurerm_kubernetes_cluster" "example" {
  name                = "multi-cluster"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  dns_prefix          = "multi-cluster"
  default_node_pool {
    name       = "default"
    node_count = 3
    vm_size    = "Standard_DS2_v2"
  }
}
```

3. Connect Rancher to Azure AKS
    - Install and configure Rancher then add AKS clusters for multi-cluster management.

---

3. Google Cloud Platform Setup (with GKE)

Pre-requisites

- Google Cloud account with sufficient permissions.
- GCP CLI (gcloud) configured.
- Terraform installed.
- kubectl installed.

Steps

1. **Create GKE Cluster**
    - Define the GKE cluster configuration.

```
resource "google_container_cluster" "example" {
  name     = "multi-cluster"
  location = "us-central1-a"
  initial_node_count = 3
  node_config {
    machine_type = "n1-standard-2"
  }
}
```

2. **Create Kubernetes Credentials**
    - Use the google_container_cluster output to fetch kubeconfig and authenticate your kubectl tool.

```
output "kube_config" {
```

```
  value = google_container_cluster.example.kube_config[0].raw_kube_config
}
```

3. **Connect Rancher to GKE**
    - Install Rancher and register your GKE cluster for central management.

---

4. Oracle Cloud Setup (with OKE)

Pre-requisites

- Oracle Cloud account with sufficient permissions.
- OCI CLI configured.
- Terraform installed.
- kubectl installed.

Steps

1. Create a VCN and Subnets
    - Define the network configuration for your Kubernetes clusters.

```
resource "oci_core_virtual_network" "example" {
  compartment_id = var.compartment_id
  display_name   = "multi-cluster-vcn"
  cidr_block     = "10.0.0.0/16"
}
```

2. **Create OKE Cluster**
    - Define the OKE cluster in Terraform.

```
resource "oci_containerengine_cluster" "example" {
  compartment_id = var.compartment_id
```

```
  name          = "multi-cluster"

  vcn_id        = oci_core_virtual_network.example.id

}
```

3. **Connect Rancher to Oracle OKE**
    - Set up Rancher and register your Oracle Kubernetes Engine (OKE) cluster for centralized management.

General Integration with Rancher

1. **Install Rancher**
    - You can install Rancher on a VM or Kubernetes cluster to serve as your multi-cluster management interface.

```
docker run -d --restart=unless-stopped -p 808080 rancher/rancherlatest
```

2. **Add Clusters to Rancher**
    - After provisioning the Kubernetes clusters add them to Rancher for centralized management using Rancher's UI or CLI.

```
kubectl config use-context <cluster-name>
```

3. **Monitor and Manage Clusters**
    - Use Rancher's dashboard to view configure and monitor all your Kubernetes clusters across AWS Azure GCP and Oracle Cloud.

Conclusion

By using Terraform you can easily create and manage Kubernetes clusters across multiple cloud providers. This project demonstrates how you can leverage tools like Rancher for multi-cluster management allowing you to manage clusters seamlessly across different cloud environments. With centralized management you can focus more on deploying and scaling your applications instead of managing the individual clusters.

Project 5 Terraform for Kubernetes Multi-Cluster Management

Project Overview

This project involves provisioning and managing multiple Kubernetes clusters across different cloud providers (AWS Azure GCP Oracle Cloud) using Terraform. By integrating with tools like Rancher we can centralize the management of these clusters making it easier to handle multi-region or multi-cloud environments. Terraform's Infrastructure as Code (IaC) approach allows for reproducible declarative configurations ensuring scalability and consistency across clusters.

---

1. AWS Setup (with EKS)

Pre-requisites

- AWS account with sufficient permissions.
- AWS CLI configured.
- Terraform installed.
- `kubectl` and `eksctl` installed.

Steps

Create IAM Role and Policies for EKS

```
resource "aws_iam_role" "eks_cluster_role" {

  name = "eks-cluster-role"

  assume_role_policy = jsonencode({

    Version = "2012-10-17"

    Statement = [

      {

        Action = "sts:AssumeRole"

        Effect = "Allow"
```

```
        Principal = {
          Service = "eks.amazonaws.com"
        }
      }
    ]
  })
}
```

Create EKS Cluster

```
resource "aws_eks_cluster" "example" {
  name     = "multi-cluster"
  role_arn = aws_iam_role.eks_cluster_role.arn
  vpc_config {
    subnet_ids = aws_subnet.subnet_ids[*].id
  }
}
```

Create Worker Nodes

```
resource "aws_eks_node_group" "example" {
  cluster_name    = aws_eks_cluster.example.name
  node_group_name = "example-node-group"
  node_role_arn   = aws_iam_role.eks_node_group_role.arn
```

```
  subnet_ids      = aws_subnet.subnet_ids[*].id
}
```

Install Rancher

Once the EKS cluster is created, install Rancher to manage the cluster.

Steps to Install Rancher

- Run Rancher in a Docker container or deploy it to the EKS cluster:

```
docker run -d --restart=unless-stopped -p 8080:80
rancher/rancher:latest
```

---

2. Azure Setup (with AKS)

Pre-requisites

- Azure account with sufficient permissions.
- Azure CLI configured.
- Terraform installed.
- `kubectl` installed.

Steps

Create a Resource Group

```
resource "azurerm_resource_group" "example" {
  name     = "multi-cluster-rg"
  location = "East US"
}
```

Create AKS Cluster

```
resource "azurerm_kubernetes_cluster" "example" {
  name                = "multi-cluster"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  dns_prefix          = "multi-cluster"
  default_node_pool {
    name       = "default"
    node_count = 3
    vm_size    = "Standard_DS2_v2"
  }
}
```

Install Rancher

After creating the AKS cluster, install Rancher if it wasn't installed during AWS setup. Alternatively, you can add the Azure cluster to an existing Rancher instance.

Steps to Install Rancher

- Use the following Docker command or deploy Rancher onto the AKS cluster:

```
docker run -d --restart=unless-stopped -p 8080:80
rancher/rancher:latest
```

---

3. Google Cloud Platform Setup (with GKE)

Pre-requisites

- Google Cloud account with sufficient permissions.
- GCP CLI (`gcloud`) configured.
- Terraform installed.
- `kubectl` installed.

Steps

Create GKE Cluster

```
resource "google_container_cluster" "example" {

  name               = "multi-cluster"

  location           = "us-central1-a"

  initial_node_count = 3

  node_config {

    machine_type = "n1-standard-2"

  }

}
```

Create Kubernetes Credentials

```
output "kube_config" {

  value =
google_container_cluster.example.kube_config[0].raw_kube_config

}
```

Install Rancher

Install Rancher if needed, or add the GKE cluster to an existing Rancher instance.

Steps to Install Rancher

- Deploy Rancher using Docker or onto the GKE cluster:

```
docker run -d --restart=unless-stopped -p 8080:80
rancher/rancher:latest
```

---

4. Oracle Cloud Setup (with OKE)

Pre-requisites

- Oracle Cloud account with sufficient permissions.
- OCI CLI configured.
- Terraform installed.
- `kubectl` installed.

Steps

Create a VCN and Subnets

```
resource "oci_core_virtual_network" "example" {

  compartment_id = var.compartment_id

  display_name   = "multi-cluster-vcn"

  cidr_block     = "10.0.0.0/16"

}
```

Create OKE Cluster

```
resource "oci_containerengine_cluster" "example" {

  compartment_id = var.compartment_id

  name           = "multi-cluster"

  vcn_id         = oci_core_virtual_network.example.id

}
```

Install Rancher

If Rancher is not already installed, deploy it to manage your OKE cluster.

Steps to Install Rancher

- Use the Docker command or deploy Rancher on the OKE cluster:

```
docker run -d --restart=unless-stopped -p 8080:80
rancher/rancher:latest
```

Rancher: General Integration for All Clusters

Once Rancher is installed (on AWS, Azure, GCP, or Oracle Cloud), add all Kubernetes clusters to Rancher.

Steps to Add Clusters to Rancher

- Open Rancher UI: `http://<Rancher_IP>:8080`.
- Use Rancher's interface to register clusters:
    - For EKS, AKS, GKE, or OKE clusters, follow Rancher's guided cluster import process.
- Use `kubectl` to switch between cluster contexts:

```
kubectl config use-context <cluster-name>
```

Conclusion

By integrating Rancher with Terraform-managed Kubernetes clusters across AWS, Azure, GCP, and Oracle Cloud, you achieve centralized, streamlined multi-cluster management. Rancher ensures that monitoring, scaling, and configuration management are simplified across all environments.

Project 6 Terraform for Kubernetes with Istio Service Mesh

This project involves setting up a Kubernetes cluster using Terraform and configuring Istio as a service mesh for managing traffic enforcing security policies and enabling observability features like tracing and monitoring. Terraform will be used to automate the infrastructure provisioning on cloud platforms like AWS Azure GCP and Oracle Cloud with the Kubernetes cluster deployed and Istio installed to manage microservices traffic.

Prerequisites

- **Terraform** installed on your local machine.
- **Cloud account** (AWS Azure GCP Oracle Cloud).
- **kubectl** installed and configured for your local machine.
- **Helm** installed for deploying Istio.
- Basic knowledge of Kubernetes and Terraform.

Steps for AWS

Set up AWS Provider
In your main.tf configure the AWS provider to allow Terraform to interact with your AWS environment.

```
provider "aws" {

  region = "us-west-2"

}
```

Create VPC and Subnets
Define the VPC and subnets for your Kubernetes cluster.

```
resource "aws_vpc" "k8s_vpc" {

  cidr_block = "10.0.0.0/16"

}
```

```
resource "aws_subnet" "k8s_subnet" {
```

```
  vpc_id                  = aws_vpc.k8s_vpc.id
  cidr_block              = "10.0.1.0/24"
  availability_zone       = "us-west-2a"
  map_public_ip_on_launch = true
}
```

Create EC2 Instances for Kubernetes Nodes
Provision EC2 instances to serve as Kubernetes nodes.

```
resource "aws_instance" "k8s_nodes" {
  count           = 3
  ami             = "ami-0abcdef1234567890"
  instance_type   = "t2.medium"
  subnet_id       = aws_subnet.k8s_subnet.id
  key_name        = "your-ssh-key"
  associate_public_ip_address = true
}
```

Install Kubernetes with Kubeadm
Use the EC2 instances to manually set up Kubernetes using kubeadm or use a managed service like **EKS** if you prefer.

Install Istio using Helm
Once your cluster is set up install Istio.

```
helm repo add istio https//istio-release.storage.googleapis.com/charts
helm install istio-base istio/istio-base
helm install istiod istio/istiod
```

helm install istio-ingress istio/istio-ingressgateway

Configure Istio for Traffic Management
After Istio is installed configure ingress and egress rules as well as service policies and security features.

Steps for Azure

Set up Azure Provider
In your main.tf configure the Azure provider.

```
provider "azurerm" {
  features {}
}
```

Create Azure Virtual Network and Subnets
Define the Virtual Network and Subnets for the Kubernetes cluster.

```
resource "azurerm_virtual_network" "k8s_vnet" {
  name                = "k8s-vnet"
  address_space       = ["10.0.0.0/16"]
  location            = "East US"
  resource_group_name = azurerm_resource_group.rg.name
}

resource "azurerm_subnet" "k8s_subnet" {
  name                 = "k8s-subnet"
  resource_group_name  = azurerm_resource_group.rg.name
```

```
  virtual_network_name = azurerm_virtual_network.k8s_vnet.name
  address_prefixes     = ["10.0.1.0/24"]
}
```

Create Azure Kubernetes Service (AKS)
Deploy AKS to handle the Kubernetes setup automatically.

```
resource "azurerm_kubernetes_cluster" "aks_cluster" {
  name                = "aks-cluster"
  location            = "East US"
  resource_group_name = azurerm_resource_group.rg.name
  default_node_pool {
    name       = "default"
    node_count = 3
    vm_size    = "Standard_DS2_v2"
  }
  identity {
    type = "SystemAssigned"
  }
}
```

Install Istio on AKS
After your AKS cluster is provisioned use Helm to install Istio as in AWS setup.

Steps for GCP

Set up Google Cloud Provider
Configure the GCP provider in your main.tf.

```
provider "google" {
  project = "your-project-id"
  region  = "us-central1"
}
```

Create VPC and Subnets
Define the VPC and subnet for your Kubernetes cluster.

```
resource "google_compute_network" "k8s_network" {
  name = "k8s-network"
  auto_create_subnetworks = "true"
}
```

Create Google Kubernetes Engine (GKE) Cluster
Use GKE for automatic Kubernetes cluster provisioning.

```
resource "google_container_cluster" "gke_cluster" {
  name     = "gke-cluster"
  location = "us-central1-a"
  initial_node_count = 3
  node_config {
    machine_type = "e2-medium"
  }
```

```
}
```

1. **Install Istio on GKE**
   Install Istio using Helm as in previous steps.

Oracle Cloud

Set up Oracle Cloud Provider
Configure the Oracle Cloud provider in your main.tf.

```
provider "oci" {
  tenancy_ocid        = "your-tenancy-id"
  user_ocid           = "your-user-id"
  fingerprint         = "your-fingerprint"
  private_key_path    = "path/to/private_key"
  region              = "us-phoenix-1"
}
```

Create Oracle Cloud Virtual Network
Define a VCN and subnets for your Kubernetes cluster.

```
resource "oci_core_virtual_network" "k8s_vcn" {
  compartment_id = "your-compartment-id"
  display_name   = "k8s-vcn"
  cidr_block     = "10.0.0.0/16"
}
```

Create Oracle Cloud Container Engine for Kubernetes (OKE)
Use Oracle's managed Kubernetes service to deploy OKE.

```
resource "oci_containerengine_cluster" "oke_cluster" {
  compartment_id = "your-compartment-id"
  name           = "oke-cluster"
  vcn_id         = oci_core_virtual_network.k8s_vcn.id
  kubernetes_version = "v1.19.7"
}
```

Install Istio on OKE
Install Istio using Helm.

Final Considerations for All Clouds

- **Traffic Management** Istio allows you to configure traffic routing retries timeouts and load balancing at the service level.
- **Security** Istio provides mTLS for secure communication between microservices enforcing encryption and authentication policies.
- **Observability** Istio integrates with monitoring tools like Prometheus and Grafana providing insights into the health and performance of services.

By following these steps you'll be able to provision a cloud-native Kubernetes cluster on your preferred platform (AWS Azure GCP or Oracle) and configure Istio for managing traffic security and observability.

---

6. CI/CD Pipeline

Project 1. CI/CD Pipeline for Infrastructure

Create a Terraform-based pipeline that automatically provisions and updates infrastructure when changes are pushed to the code repository.

To create a CI/CD pipeline for provisioning and updating infrastructure using Terraform across multiple cloud platforms (AWS Azure GCP and Oracle Cloud) follow these steps

---

Project Overview

The goal is to build a Terraform-based CI/CD pipeline that triggers on repository changes and deploys infrastructure to the respective cloud platform. This is achieved by setting up separate workflows for AWS Azure GCP and Oracle Cloud.

Directory Structure

plaintext

```
terraform-infra-pipeline/
├── aws/
│   ├── main.tf
│   ├── variables.tf
│   ├── outputs.tf
│   └── terraform.tfvars
├── azure/
│   ├── main.tf
│   ├── variables.tf
```

```
|   ├── outputs.tf
|   └── terraform.tfvars
├── gcp/
|   ├── main.tf
|   ├── variables.tf
|   ├── outputs.tf
|   └── terraform.tfvars
├── oracle/
|   ├── main.tf
|   ├── variables.tf
|   ├── outputs.tf
|   └── terraform.tfvars
├── Jenkinsfile
└── README.md
```

Step 1 Configure Terraform for Each Cloud Provider

1. AWS

Create `main.tf` to define resources (e.g. EC2 instances S3 buckets).

Configure the provider

```
provider "aws" {
```

```
  region = var.region
}

resource "aws_instance" "example" {
  ami           = var.ami
  instance_type = var.instance_type
}
```

Define `variables.tf` for inputs

```
variable "region" {}
variable "ami" {}
variable "instance_type" {}
```

Create `terraform.tfvars` for default values

```
region         = "us-east-1"
ami            = "ami-0c55b159cbfafe1f0"
instance_type  = "t2.micro"
```

---

2. Azure

Create `main.tf` for Azure resources

```
provider "azurerm" {
  features = {}
}

resource "azurerm_resource_group" "example" {
  name     = var.resource_group_name
  location = var.location
}
```

Define `variables.tf`

```
variable "resource_group_name" {}
variable "location" {}
```

Add `terraform.tfvars`

```
resource_group_name = "example-rg"
location            = "East US"
```

---

3. GCP

Create `main.tf`

```
provider "google" {
  project = var.project
  region  = var.region
}

resource "google_compute_instance" "example" {
  name         = "example-instance"
  machine_type = "e2-micro"
  zone         = var.zone

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-11"
    }
  }

  network_interface {
    network = "default"
  }
}
```

Define `variables.tf`

```
variable "project" {}
variable "region" {}
variable "zone" {}
```

- Add `terraform.tfvars`

```
project = "my-gcp-project"
region  = "us-central1"
zone    = "us-central1-a"
```

---

4. Oracle

Create `main.tf`

```
provider "oci" {
  tenancy_ocid     = var.tenancy_ocid
  user_ocid        = var.user_ocid
  fingerprint      = var.fingerprint
  private_key_path = var.private_key_path
  region           = var.region
```

```
}


resource "oci_core_instance" "example" {
  availability_domain = var.availability_domain
  compartment_id      = var.compartment_id
  shape               = "VM.Standard2.1"

  create_vnic_details {
    subnet_id = var.subnet_id
  }

  source_details {
    source_type = "image"
    source_id   = var.image_id
  }
}
```

Define `variables.tf`

```
variable "tenancy_ocid" {}
variable "user_ocid" {}
variable "fingerprint" {}
```

```
variable "private_key_path" {}

variable "region" {}

variable "availability_domain" {}

variable "compartment_id" {}

variable "subnet_id" {}

variable "image_id" {}
```

Add `terraform.tfvars`

```
tenancy_ocid     = "ocid1.tenancy.oc1..example"

user_ocid        = "ocid1.user.oc1..example"

fingerprint      = "xxxxxxxxxx"

private_key_path = "/path/to/private-key.pem"

region           = "us-ashburn-1"

availability_domain = "UocmPHX-AD-1"

compartment_id   = "ocid1.compartment.oc1..example"

subnet_id        = "ocid1.subnet.oc1..example"

image_id         = "ocid1.image.oc1..example"
```

-

Step 2 CI/CD Pipeline Configuration

Jenkinsfile

groovy

```
pipeline {
    agent any
    environment {
        TF_VERSION = '1.6.0'
    }
    stages {
        stage('Initialize Terraform') {
            steps {
                script {
                    sh '''
                    # Initialize Terraform
                    terraform -chdir=${env.TF_DIR} init
                    '''
                }
            }
        }

        stage('Validate Terraform') {
            steps {
                script {
```

```
                    sh '''
                    # Validate Terraform configurations
                    terraform -chdir=${env.TF_DIR} validate
                    '''
                }
            }
        }

        stage('Plan Terraform Changes') {
            steps {
                script {
                    sh '''
                    # Create Terraform plan
                    terraform -chdir=${env.TF_DIR} plan
-out=tfplan
                    '''
                }
            }
        }

        stage('Apply Terraform Changes') {
            steps {
```

```
                script {
                    sh '''
                    # Apply Terraform changes
                    terraform -chdir=${env.TF_DIR} apply
-auto-approve tfplan
                    '''
                }
            }
        }
    }

    post {
        always {
            cleanWs()
        }
    }
}
```

Environment-Specific Configuration

Pass the appropriate directory for the cloud provider in the pipeline

- For AWS `TF_DIR=aws/main.tf or variables.tf`
- For Azure `TF_DIR=azure/`
- For GCP `TF_DIR=gcp/`

- For Oracle `TF_DIR=oracle/`

Step 3 Trigger Pipeline

1. Push changes to the repository.
2. Configure Jenkins to poll the repository or use webhooks for automatic triggering.

Project 2. Terraform for CI/CD Tools

1. AWS - Terraform for CI/CD Tools

Step 1 Install Terraform

- Install Terraform on your local machine or CI server. You can download it from Terraform's website.

Step 2 AWS Credentials

- Set up AWS credentials on your machine
    - Create a new IAM user in the AWS Console with appropriate permissions (like AdministratorAccess for simplicity).
    - Configure AWS CLI using the aws configure command or manually set environment variables (AWS_ACCESS_KEY_ID AWS_SECRET_ACCESS_KEY).

Step 3 Define AWS Provider

Create a main.tf file with the AWS provider setup.

```
provider "aws" {

  region = "us-west-2"  # Use your desired AWS region

}
```

Step 4 Set Up Resources

- Create resources like EC2 instances or an S3 bucket for storing your CI/CD artifacts.

Example Provision an EC2 instance for Jenkins.

```
resource "aws_instance" "jenkins" {
  ami           = "ami-0c55b159cbfafe1f0"  # Use a Jenkins pre-installed AMI or any AMI
  instance_type = "t2.micro"
  key_name      = "your-key-name"
  security_groups = ["your-security-group"]

  tags = {
    Name = "Jenkins-Server"
  }
}
```

Step 5 Apply the Configuration

- Initialize and apply the Terraform configuration.

```
terraform init
terraform apply
```

---

2. Azure - Terraform for CI/CD Tools

Step 1 Install Terraform

- Same as AWS download and install Terraform.

Step 2 Set Up Azure Credentials

- Use Azure CLI to authenticate
    - az login to authenticate with Azure.
    - Configure your credentials with az account set --subscription <your-subscription-id>.

Step 3 Define Azure Provider

Create a main.tf file with the Azure provider setup.

```
provider "azurerm" {
  features {}
}
```

Step 4 Create Resources

- Example Provision an Azure Virtual Machine to run Jenkins.

```
resource "azurerm_virtual_machine" "jenkins" {
  name                = "jenkins-vm"
  location            = "East US"
  resource_group_name   = "ci-cd-rg"
  size                = "Standard_B1s"
  network_interface_ids = [azurerm_network_interface.nic.id]
  os_profile {
    computer_name  = "jenkins-vm"
    admin_username = "adminuser"
    admin_password = "adminpassword"
  }
  os_profile_linux_config {
```

```
    disable_password_authentication = false
  }
}
resource "azurerm_network_interface" "nic" {
  name                = "jenkins-nic"
  location            = "East US"
  resource_group_name = "ci-cd-rg"
  ip_configuration {
    name                          = "internal"
    subnet_id                     = azurerm_subnet.subnet.id
    private_ip_address_allocation = "Dynamic"
  }
}
```

Step 5 Apply the Configuration

- Run the following commands to apply the configuration

```
terraform init
terraform apply
```

---

3. GCP - Terraform for CI/CD Tools

Step 1 Install Terraform

- Same as previous steps download and install Terraform.

Step 2 Set Up GCP Credentials

- Authenticate via gcloud CLI
    - Install gcloud and authenticate gcloud auth login.
    - Set your project with gcloud config set project <your-project-id>.

Step 3 Define GCP Provider

Create a main.tf file with the GCP provider setup.

```
provider "google" {
  credentials = file("<path-to-your-service-account-json>")
  project     = "<your-project-id>"
  region      = "us-central1"
}
```

Step 4 Create Resources

- Example Provision a Google Compute Engine instance for Jenkins.

```
resource "google_compute_instance" "jenkins" {
  name         = "jenkins-instance"
  machine_type = "f1-micro"
  zone         = "us-central1-a"
  tags         = ["jenkins"]

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-9"
    }
```

```
  }

  network_interface {
    network = "default"
    access_config {}
  }
}
```

Step 5 Apply the Configuration

- Run the following commands to apply the configuration

```
terraform init
terraform apply
```

---

4. Oracle Cloud - Terraform for CI/CD Tools

Step 1 Install Terraform

- Download and install Terraform from the official site.

Step 2 Set Up Oracle Cloud Credentials

- Configure OCI credentials by following the OCI setup guide.

Step 3 Define Oracle Cloud Provider

Create a main.tf file with the Oracle Cloud provider setup.

```
provider "oci" {
  tenancy_ocid       = "<tenancy_ocid>"
  user_ocid          = "<user_ocid>"
  fingerprint        = "<fingerprint>"
  private_key_path   = "<private_key_path>"
  region             = "<region>"
}
```

Step 4 Create Resources

- Example Provision an Oracle Compute instance.

```
resource "oci_core_instance" "jenkins" {
  availability_domain = "UocmPHX-AD-1"
  compartment_id      = "<compartment_ocid>"
  shape               = "VM.Standard2.1"
  display_name        = "jenkins-instance"

  source_details {
    source_type = "image"
    image_id    = "<your-image-id>"
  }

  metadata = {
    ssh_authorized_keys = "<your-ssh-public-key>"
```

```
  }
  create_vnic_details {
    subnet_id = "<subnet_id>"
    assign_public_ip = true
  }
}
```

Step 5 Apply the Configuration

- Run the following commands to apply the configuration

terraform init

terraform apply

Conclusion

Project 3: GitOps with Terraform and ArgoCD

Set up GitOps practices using Terraform to provision infrastructure and ArgoCD to manage application deployments automatically.

General Prerequisites

Install Terraform
Ensure that Terraform is installed on your system.

- Download Terraform

Verify the installation by running:

```
terraform -v
```

Install ArgoCD
Install ArgoCD on your Kubernetes cluster.

- Follow installation instructions specific to your environment.

Verify the installation by running:

```
kubectl get pods -n argocd
```

GitHub/GitLab Repository
You need a Git repository to store your Terraform and ArgoCD configurations.

1. GitOps with Terraform and ArgoCD on AWS

Step 1: Set up AWS Infrastructure with Terraform

Create a Terraform configuration (`main.tf`) for provisioning AWS resources:

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_instance" "example" {
```

```
  ami           = "ami-0c55b159cbfafe1f0"  # Example AMI ID
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.main.id
}
```

Initialize Terraform:

```
terraform init
```

Apply Terraform Configuration:

```
terraform apply
```

Step 2: Set up ArgoCD for AWS Deployment

Create an ArgoCD application YAML file (`argo-app.yaml`):

yaml

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: example-app
  namespace: argocd
spec:
  destination:
    namespace: default
```

```
    server: 'https://kubernetes.default.svc'
  source:
    repoURL: 'https://github.com/yourrepo/terraform-project'
    targetRevision: HEAD
    path: './kubernetes'
  project: default
```

Deploy the ArgoCD application:

```
kubectl apply -f argo-app.yaml
```

Access ArgoCD UI to monitor deployments:

```
kubectl port-forward svc/argocd-server -n argocd 8080:80
```

---

2. GitOps with Terraform and ArgoCD on Azure

Step 1: Set up Azure Infrastructure with Terraform

Create a Terraform configuration (`main.tf`):

```
provider "azurerm" {
  features {}
}


resource "azurerm_resource_group" "main" {
  name     = "example-resources"
```

```
  location = "East US"
}

resource "azurerm_virtual_network" "main" {
  name                = "example-vnet"
  address_space       = ["10.0.0.0/16"]
  location            = azurerm_resource_group.main.location
  resource_group_name = azurerm_resource_group.main.name
}
```

Initialize Terraform:

```
terraform init
```

Apply Terraform Configuration:

```
terraform apply
```

Step 2: Set up ArgoCD for Azure Deployment

Create an ArgoCD application YAML file (`argo-app.yaml`):

yaml

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: example-app
```

```
  namespace: argocd
spec:
  destination:
    namespace: default
    server: 'https://kubernetes.default.svc'
  source:
    repoURL: 'https://github.com/yourrepo/terraform-project'
    targetRevision: HEAD
    path: './kubernetes'
  project: default
```

Deploy the ArgoCD application:

```
kubectl apply -f argo-app.yaml
```

---

3. GitOps with Terraform and ArgoCD on GCP

Step 1: Set up GCP Infrastructure with Terraform

Create a Terraform configuration (`main.tf`):

```
provider "google" {
  credentials = file("<YOUR-CREDENTIALS-FILE>.json")
  project     = "<YOUR-PROJECT-ID>"
  region      = "us-central1"
}
```

```
resource "google_compute_network" "main" {
  name                    = "example-network"
  auto_create_subnetworks  = true
}

resource "google_compute_instance" "example" {
  name         = "example-instance"
  machine_type = "f1-micro"
  zone         = "us-central1-a"
  boot_disk {
    initialize_params {
      image = "debian-9-stretch-v20191210"
    }
  }
}
```

Initialize Terraform:

```
terraform init
```

Apply Terraform Configuration:

```
terraform apply
```

Step 2: Set up ArgoCD for GCP Deployment

Create an ArgoCD application YAML file (`argo-app.yaml`):

yaml

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: example-app
  namespace: argocd
spec:
  destination:
    namespace: default
    server: 'https://kubernetes.default.svc'
  source:
    repoURL: 'https://github.com/yourrepo/terraform-project'
    targetRevision: HEAD
    path: './kubernetes'
  project: default
```

Deploy the ArgoCD application:

```
kubectl apply -f argo-app.yaml
```

---

4. GitOps with Terraform and ArgoCD on Oracle Cloud

Step 1: Set up Oracle Cloud Infrastructure (OCI) with Terraform

Create a Terraform configuration (`main.tf`):

```
provider "oci" {
  tenancy_ocid     = "<TENANCY_OCID>"
  user_ocid        = "<USER_OCID>"
  fingerprint      = "<FINGERPRINT>"
  private_key_path = "<PRIVATE_KEY_PATH>"
  region           = "us-phoenix-1"
}

resource "oci_core_virtual_network" "main" {
  compartment_id = "<COMPARTMENT_OCID>"
  cidr_block     = "10.0.0.0/16"
  display_name   = "example-vcn"
}

resource "oci_core_instance" "example" {
  compartment_id       = "<COMPARTMENT_OCID>"
  availability_domain  = "UocmPHX-AD-1"
  shape                = "VM.Standard.E2.1.Micro"
  display_name         = "example-instance"
```

```
}
```

Initialize Terraform:

```
terraform init
```

Apply Terraform Configuration:

```
terraform apply
```

Step 2: Set up ArgoCD for Oracle Cloud Deployment

Create an ArgoCD application YAML file (`argo-app.yaml`):

yaml

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: example-app
  namespace: argocd
spec:
  destination:
    namespace: default
    server: 'https://kubernetes.default.svc'
  source:
    repoURL: 'https://github.com/yourrepo/terraform-project'
    targetRevision: HEAD
```

```
    path: './kubernetes'

  project: default
```

Deploy the ArgoCD application:

```
kubectl apply -f argo-app.yaml
```

Final Notes

- **GitOps Workflow:** Once the infrastructure is set up with Terraform and the ArgoCD application is deployed, any changes to the Terraform configuration or Kubernetes manifests in the Git repository will trigger ArgoCD to automatically deploy and sync those changes to the respective cloud platform.
- **Monitoring:** Use the ArgoCD dashboard to monitor the health and status of your deployments.

Project 4 Terraform for Serverless CI/CD Pipeline

Goal Set up a serverless CI/CD pipeline using Terraform integrating tools like GitLab CI AWS CodePipeline and AWS Lambda for seamless automated deployment. This project allows for continuous deployment without managing servers leveraging the scalability of serverless architecture.

Creating a **Serverless CI/CD Pipeline** using Terraform for AWS Azure GCP and Oracle involves setting up infrastructure as code to provision cloud resources. The infrastructure components will include services like AWS Lambda Azure Functions Google Cloud Functions or Oracle Functions along with other components like version control systems build services and deployment pipelines.

1. AWS Serverless CI/CD Pipeline with Terraform

In AWS you will use AWS Lambda for the serverless compute service CodeCommit (for version control) CodePipeline (for CI/CD) and CodeBuild (for build).

Steps

Initialize Terraform Project Create a directory and initialize your Terraform project.

```
mkdir aws-serverless-cicd

cd aws-serverless-cicd

terraform init
```

Set Up AWS Lambda Create a Lambda function using Terraform

```
resource "aws_lambda_function" "example" {
  function_name = "my-function"
  runtime       = "nodejs14.x"
  handler       = "index.handler"
  role          = aws_iam_role.lambda_exec.arn
  filename      = "function.zip"
}
```

Set Up CodeCommit (for Version Control)

```
resource "aws_codecommit_repository" "repo" {
  repository_name = "my-repo"
}
```

Set Up CodeBuild (for Build)

```
resource "aws_codebuild_project" "build" {
  name          = "my-build"
  description   = "Build project"
  service_role  = aws_iam_role.codebuild_exec.arn
```

```
  artifacts      = { type = "NO_ARTIFACTS" }
  environment {
    compute_type = "BUILD_GENERAL1_SMALL"
    image        = "aws/codebuild/standard4.0"
    type         = "LINUX_CONTAINER"
  }
}
```

Set Up CodePipeline (CI/CD)

```
resource "aws_codepipeline" "ci_cd_pipeline" {
  name     = "ci-cd-pipeline"
  role_arn = aws_iam_role.codepipeline_exec.arn

  artifact_store {
    location = aws_s3_bucket.artifact_store.bucket
    type     = "S3"
  }

  stage {
    name = "Source"
    action {
      name          = "Source"
      category      = "Source"
      owner         = "AWS"
      provider      = "CodeCommit"
```

```
    version       = "1"
    output_artifacts = ["source_output"]
    configuration = {
      RepositoryName = aws_codecommit_repository.repo.repository_name
      BranchName     = "main"
    }
  }
}

stage {
  name = "Build"
  action {
    name          = "Build"
    category      = "Build"
    owner         = "AWS"
    provider      = "CodeBuild"
    version       = "1"
    input_artifacts = ["source_output"]
    configuration = {
      ProjectName = aws_codebuild_project.build.name
    }
  }
}
```

```
  stage {
    name = "Deploy"
    action {
      name          = "Deploy"
      category      = "Invoke"
      owner         = "AWS"
      provider      = "Lambda"
      version       = "1"
      input_artifacts = ["build_output"]
      configuration = {
        FunctionName = aws_lambda_function.example.function_name
      }
    }
  }
}
```

Deploy Run Terraform to apply the configuration

terraform apply

---

2. Azure Serverless CI/CD Pipeline with Terraform

In Azure you will use Azure Functions for the serverless compute service Azure DevOps or GitHub for version control Azure Pipelines for CI/CD and Azure Functions for deployment.

Steps

Initialize Terraform Project

mkdir azure-serverless-cicd

cd azure-serverless-cicd

terraform init

Set Up Azure Function App

```
resource "azurerm_function_app" "example" {
  name                = "my-function-app"
  location            = "East US"
  resource_group_name = azurerm_resource_group.rg.name
  app_service_plan_id = azurerm_app_service_plan.plan.id
  storage_connection_string = azurerm_storage_account.storage.primary_connection_string
}
```

Set Up Azure DevOps Project You can use the Terraform Azure DevOps provider to configure your pipelines

```
resource "azuredevops_project" "example" {
  name            = "my-project"
  visibility      = "private"
  version_control  = "Git"
  work_item_template = "Agile"
}
```

Set Up Azure Pipelines Create a pipeline to build and deploy your function.

```
resource "azuredevops_build_definition" "build" {
  project_id = azuredevops_project.example.id
  name       = "Build Function"

  repository {
    url = "https//dev.azure.com/your_org/your_project/_git/your_repo"
    type = "Git"
  }

  ci_trigger {
    use_yaml = true
    yaml_file = "azure-pipelines.yml"
  }
}
```

Deploy

```
terraform apply
```

---

3. GCP Serverless CI/CD Pipeline with Terraform

In Google Cloud you'll use Google Cloud Functions for serverless compute Cloud Source Repositories for version control Cloud Build for build and Cloud Deploy for deployment.

Steps

Initialize Terraform Project

mkdir gcp-serverless-cicd

cd gcp-serverless-cicd

terraform init

Set Up Google Cloud Function

```
resource "google_cloudfunctions_function" "example" {
  name        = "my-function"
  runtime     = "nodejs16"
  entry_point = "helloWorld"
  available_memory_mb = 256
  source_archive_bucket = google_storage_bucket.bucket.name
  source_archive_object = google_storage_bucket_object.object.name
}
```

Set Up Cloud Source Repository

```
resource "google_sourcerepo_repository" "repo" {
  name = "my-repo"
}
```

Set Up Cloud Build

```
resource "google_cloudbuild_trigger" "trigger" {

  name = "build-trigger"


  github {

    owner = "your-github-owner"

    name  = "your-repo"

  }


  build {

    step {

      name = "gcr.io/cloud-builders/gcloud"

      args = ["builds" "submit"]

    }

  }

}
```

Deploy

```
terraform apply
```

---

4. Oracle Cloud Serverless CI/CD Pipeline with Terraform

In Oracle Cloud you will use Oracle Functions for serverless compute Oracle Cloud Infrastructure (OCI) for version control and deployment automation.

Steps

Initialize Terraform Project

mkdir oracle-serverless-cicd

cd oracle-serverless-cicd

terraform init

Set Up Oracle Function

```
resource "oci_functions_function" "example" {
  display_name = "my-function"
  application_id = oci_functions_application.app.id
  image = "your-function-image"
  memory_in_mbs = 256
}
```

Set Up Oracle Cloud Infrastructure Registry (OCIR) Store your Docker image in OCIR and deploy to functions.

```
resource "oci_artifacts_container_image" "function_image" {
  repository_name = "my-repo"
  compartment_id  = var.compartment_id
  image_tag       = "latest"
  image_digest    = "your-image-digest"
}
```

Set Up Oracle Cloud Build Use the OCI DevOps service for CI/CD pipelines.

```
resource "oci_devops_project" "devops_project" {
  display_name = "my-devops-project"
  compartment_id = var.compartment_id
}

resource "oci_devops_build_pipeline" "build_pipeline" {
  project_id = oci_devops_project.devops_project.id
  display_name = "my-build-pipeline"
}
```

Deploy
terraform apply

Project 5: Terraform with Ansible Integration

Combine **Terraform** with **Ansible** to automate infrastructure provisioning and configuration management across AWS, Azure, GCP, and Oracle Cloud. This project involves using **Terraform** to define and provision infrastructure and **Ansible** to configure the provisioned resources.

---

1. AWS Terraform + Ansible Integration

Step 1: Setup Terraform for AWS

- Install Terraform if not already installed.
- Create Terraform configuration files for provisioning AWS resources, such as EC2 instances.

provider.tf

```
provider "aws" {
  region     = "us-east-1"
  access_key = "YOUR_AWS_ACCESS_KEY"
  secret_key = "YOUR_AWS_SECRET_KEY"
}
```

main.tf

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"  # Replace with a valid AMI ID
  instance_type = "t2.micro"

  tags = {
    Name = "Terraform AWS Instance"
  }
}
```

Step 2: Setup Ansible

- Install Ansible on your local machine.
- Create an inventory file for the EC2 instance.

inventory.ini

ini

```
[aws_instances]
```

your-ec2-instance-ip

- Write an Ansible playbook to configure the instance.

configure-aws.yml

yaml

```
---
- name: Configure EC2 instance
  hosts: aws_instances
  become: yes
  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present
```

Step 3: Integrate Terraform and Ansible

- Modify main.tf to output the EC2 instance public IP.

Output block in main.tf:

```
output "instance_ip" {
  value = aws_instance.example.public_ip
}
```

- Run Terraform:

terraform init

terraform apply

- **Use Ansible to configure the provisioned instance:**

ansible-playbook -i inventory.ini configure-aws.yml

---

2. Azure Terraform + Ansible Integration

Step 1: Setup Terraform for Azure

provider.tf

```
provider "azurerm" {
  features {}
  client_id       = "YOUR_CLIENT_ID"
  client_secret   = "YOUR_CLIENT_SECRET"
  tenant_id       = "YOUR_TENANT_ID"
  subscription_id = "YOUR_SUBSCRIPTION_ID"
}
```

main.tf

```
resource "azurerm_virtual_machine" "example" {
  name                = "example-vm"
  location            = "East US"
```

```
  resource_group_name = "example-resources"
  vm_size             = "Standard_B1ms"

  storage_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "18.04-LTS"
  }

  os_profile {
    computer_name  = "example-vm"
    admin_username = "azureuser"
    admin_password = "P@ssw0rd1234"
  }

  network_interface_ids = [
    azurerm_network_interface.example.id,
  ]
}
```

Step 2: Setup Ansible

- Create an inventory file for the Azure VM.

inventory.ini

ini

[azure_vms]

azure-vm-ip

- Write an Ansible playbook for configuration.

configure-azure.yml

yaml

```
---
- name: Configure Azure VM
  hosts: azure_vms
  become: yes
  tasks:
    - name: Install Apache
      apt:
        name: apache2
        state: present
```

Step 3: Integrate Terraform and Ansible

- After provisioning the infrastructure with Terraform, run the Ansible playbook.

ansible-playbook -i inventory.ini configure-azure.yml

---

3. GCP Terraform + Ansible Integration

Step 1: Setup Terraform for GCP

provider.tf

```
provider "google" {
  credentials = file("path/to/your/credentials-file.json")
  project     = "your-project-id"
  region      = "us-central1"
}
```

main.tf

```
resource "google_compute_instance" "example" {
  name         = "example-instance"
  machine_type = "e2-micro"
  zone         = "us-central1-a"

  boot_disk {
    initialize_params {
      image = "ubuntu-os-cloud/ubuntu-2004-lts"
    }
  }

  network_interface {
```

```
    network = "default"
    access_config {
      # Include this for an external IP
    }
  }
}
```

Step 2: Setup Ansible

- Create an inventory file for the GCP instance.

inventory.ini

ini

```
[gcp_instances]
your-gcp-instance-ip
```

- Write an Ansible playbook for configuration.

configure-gcp.yml

yaml

```
---
- name: Configure GCP VM
  hosts: gcp_instances
  become: yes
  tasks:
```

```
  - name: Install Docker
    apt:
      name: docker.io
      state: present
```

Step 3: Integrate Terraform and Ansible

- After running Terraform, use Ansible to configure the instance.

```
ansible-playbook -i inventory.ini configure-gcp.yml
```

---

4. Oracle Cloud Terraform + Ansible Integration

Step 1: Setup Terraform for Oracle Cloud

provider.tf

```
provider "oci" {
  tenancy_ocid     = "your-tenancy-id"
  user_ocid        = "your-user-id"
  fingerprint      = "your-fingerprint"
  private_key_path = "path-to-private-key"
  region           = "us-phoenix-1"
}
```

main.tf

```
resource "oci_core_instance" "example" {
  availability_domain = "your-availability-domain"
  compartment_id      = "your-compartment-id"
  shape               = "VM.Standard.E2.1.Micro"

  create_vnic_details {
    subnet_id = "your-subnet-id"
  }

  source_details {
    source_type = "image"
    image_id    = "your-image-id"
  }
}
```

Step 2: Setup Ansible

- Create an inventory file for the Oracle instance.

inventory.ini

ini

```
[oracle_cloud_instances]
your-oracle-instance-ip
```

- Write an Ansible playbook for configuration.

configure-oracle.yml

yaml

```
---
- name: Configure Oracle Cloud instance
  hosts: oracle_cloud_instances
  become: yes
  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present
```

Step 3: Integrate Terraform and Ansible

- After provisioning the instance with Terraform, use Ansible to configure it.

```
ansible-playbook -i inventory.ini configure-oracle.yml
```

Conclusion

This project demonstrates how to integrate **Terraform** and **Ansible** to automate infrastructure provisioning and configuration management for **AWS**, **Azure**, **GCP**, and **Oracle Cloud**. By combining Terraform's infrastructure as code capabilities with Ansible's configuration management, you achieve a fully automated workflow across multiple cloud platforms.