Terraform Project Part-2

Terraform Project Part-2

---

7. Security & Compliance

Project 1: Bastion Host

A **Bastion Host** serves as a secure entry point to a private network, providing access to internal resources like databases and servers from the public internet.

---

1. AWS Bastion Host Configuration

Steps:

1. **Security Group:** Define a security group to allow SSH access on port 22.
2. **SSH Key Pair:** Specify your public SSH key for secure access to the Bastion Host.
3. **Instance:** Launch an EC2 instance in a public subnet with the specified security group and key pair.

```
provider "aws" {

  region = "us-west-2"

}



resource "aws_security_group" "bastion_sg" {

  name        = "bastion-sg"

  description = "Allow SSH access to Bastion Host"

  ingress {

    from_port   = 22

    to_port     = 22

    protocol    = "tcp"

    cidr_blocks = ["0.0.0.0/0"] # Replace with your IP range

  }

  egress {

    from_port   = 0

    to_port     = 0

    protocol    = "-1"

    cidr_blocks = ["0.0.0.0/0"]

  }
}

resource "aws_key_pair" "bastion_key" {
  key_name   = "bastion-key"
  public_key = file("~/.ssh/id_rsa.pub") # Replace with your public key path
}

resource "aws_instance" "bastion" {
  ami                    = "ami-0c55b159cbfafe1f0" # Replace with your AMI ID
  instance_type          = "t2.micro"
  security_groups        = [aws_security_group.bastion_sg.name]
  key_name               = aws_key_pair.bastion_key.key_name
  subnet_id              = "subnet-0bb1c79de3EXAMPLE" # Replace with your subnet ID
  associate_public_ip_address = true
  tags = {
    Name = "Bastion Host"
  }
}
```

---

2. Azure Bastion Host Configuration

Steps:

1. **Network Security Group:** Configure an NSG to allow SSH access (port 22).

2. **Network Interface:** Set up the network interface for the Bastion VM, associating it with the NSG.
3. **VM:** Launch the virtual machine using the network interface and provide SSH keys for secure access.

```
provider "azurerm" {

  features {}

}


resource "azurerm_network_security_group" "bastion_sg" {

  name                = "bastion-sg"

  location            = "West US"

  resource_group_name = "myResourceGroup"

  security_rule {

    name                       = "SSH"

    priority                   = 100

    direction                  = "Inbound"

    access                     = "Allow"

    protocol                   = "Tcp"

    source_port_range          = "*"

    destination_port_range     = "22"

    source_address_prefix      = "*"

    destination_address_prefix = "*"

  }
```

```
}

resource "azurerm_network_interface" "bastion_nic" {
  name                      = "bastion-nic"
  location                  = "West US"
  resource_group_name       = "myResourceGroup"
  network_security_group_id = azurerm_network_security_group.bastion_sg.id
  ip_configuration {
    name                          = "internal"
    subnet_id                     = "subnet_id" # Replace with your subnet ID
    private_ip_address_allocation = "Dynamic"
    public_ip_address {
      allocation_method = "Static"
    }
  }
}

resource "azurerm_linux_virtual_machine" "bastion" {
  name                = "bastion-vm"
  resource_group_name = "myResourceGroup"
  location            = "West US"
  size                = "Standard_B1s"
  network_interface_ids = [
    azurerm_network_interface.bastion_nic.id
  ]
  admin_username = "adminuser"
  admin_password = "adminpassword" # Use SSH key or managed identity instead
  ssh_keys {
    path     = "/home/adminuser/.ssh/authorized_keys"
    key_data = file("~/.ssh/id_rsa.pub") # Replace with your public key path
  }
  tags = {
    Name = "Bastion Host"
  }
}
```

---

3. GCP Bastion Host Configuration

Steps:

1. **Firewall Rule:** Create a firewall rule to allow SSH on port 22.
2. **Instance:** Launch a VM with the firewall and SSH keys configured.

```
provider "google" {
  project = "my-project-id"
  region  = "us-west1"
}
```

```
resource "google_compute_firewall" "bastion_fw" {
  name    = "bastion-fw"
  network = "default"
  allow {
    protocol = "tcp"
    ports    = ["22"]
  }
  source_ranges = ["0.0.0.0/0"] # Replace with your IP range
}

resource "google_compute_instance" "bastion" {
  name         = "bastion-vm"
  machine_type = "f1-micro"
  zone         = "us-west1-a"
  tags         = ["bastion"]
  boot_disk {
    initialize_params {
      image = "projects/debian-cloud/global/images/family/debian-10"
    }
  }
  network_interface {
    network = "default"
    access_config {}

  }

  metadata_startup_script = "#!/bin/\napt-get update && apt-get install -y ssh"

  service_account {

    scopes = ["https://www.googleapis.com/auth/cloud-platform"]

  }

  metadata = {

    ssh-keys = "your-username:${file("~/.ssh/id_rsa.pub")}" # Replace with your public key

  }

}
```

---

4. Oracle Cloud Bastion Host Configuration

Steps:

1. **Security List:** Set up a security list allowing SSH on port 22.
2. **Instance:** Launch a VM and attach the security list and SSH key for access.

```
provider "oci" {

  region = "us-phoenix-1"

}


resource "oci_core_security_list" "bastion_sg" {

  compartment_id = "compartment_ocid"
  display_name   = "bastion-sg"
  ingress_security_rules {
    protocol = "6" # TCP
    source   = "0.0.0.0/0" # Replace with your IP range
    source_type = "CIDR_BLOCK"
    tcp_options {
      min = 22
      max = 22
    }
  }
}

resource "oci_core_instance" "bastion" {
  availability_domain = "Uocm:PHX-AD-1"
  compartment_id      = "compartment_ocid"
  display_name        = "bastion-instance"
  shape               = "VM.Standard.E2.1.Micro"
  create_vnic_details {
    assign_public_ip = true
    subnet_id        = "subnet_ocid"
    display_name     = "bastion-vnic"
  }
  source_details {
    source_type = "image"

    image_id    = "image_ocid" # Replace with your image OCID

  }

  metadata = {

    ssh_authorized_keys = file("~/.ssh/id_rsa.pub") # Replace with your public key

  }

}
```

Summary:

This project demonstrates the configuration of **Bastion Hosts** on four major cloud providers: **AWS**, **Azure**, **GCP**, and **Oracle Cloud**. Each configuration includes the setup of the required security group or security list, the creation of an instance with SSH access, and using SSH keys for secure access.

Project 2. Statefile Management

Terraform state management is a critical component for tracking infrastructure changes and maintaining consistency in your cloud environments. The state file (terraform.tfstate) represents the current state of your infrastructure and is used by Terraform to plan and apply changes. By storing this state file remotely, you ensure that your infrastructure state is secure, shared, and easily accessible for teams working collaboratively. This guide explores how to configure remote state storage using cloud-native solutions such as AWS S3, Azure Blob Storage, GCP Google Cloud Storage, and Oracle Cloud Object Storage, helping you manage infrastructure more effectively across multiple cloud providers.

1. AWS Configuration: Using S3 and DynamoDB for Terraform State

Objective: Store the Terraform state file in an S3 bucket and use DynamoDB for state locking to prevent race conditions.

Steps:

- **Set up an S3 bucket for state storage**
- **Set up a DynamoDB table for state locking**

Terraform Configuration:

```
provider "aws" {

  region = "us-west-2"

}


# Create S3 bucket for state storage

resource "aws_s3_bucket" "terraform_state" {

  bucket = "your-terraform-state-bucket"

  acl    = "private"

}


# Create DynamoDB table for state locking

resource "aws_dynamodb_table" "terraform_lock" {

  name         = "terraform-lock"

  hash_key     = "LockID"

  read_capacity = 1

  write_capacity = 1

  attribute {

    name = "LockID"

    type = "S"
  }
}

# Backend configuration to use S3 and DynamoDB for state management
terraform {
  backend "s3" {
    bucket         = aws_s3_bucket.terraform_state.bucket
    key            = "state/terraform.tfstate"
    region         = "us-west-2"
    encrypt        = true
    dynamodb_table = aws_dynamodb_table.terraform_lock.name
  }
}
```

Initialize Terraform:

```
terraform init
```

---

2. Azure Configuration: Using Azure Blob Storage for Terraform State

Objective: Store the Terraform state file in Azure Blob Storage.

Steps:

- **Create a resource group**
- **Create a storage account**
- **Create a container for the state file**

Terraform Configuration:

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

# Create a resource group
resource "azurerm_resource_group" "rg" {
  name     = "terraform-rg"
  location = "East US"
}

# Create storage account for state file
resource "azurerm_storage_account" "terraform" {
  name                     = "terraformstorageaccount"
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}

# Create a container for storing the state file
```

```
resource "azurerm_storage_container" "terraform_state" {
  name                  = "terraform-state"
  storage_account_name  = azurerm_storage_account.terraform.name
  container_access_type = "private"
}

# Backend configuration to use Azure Blob Storage for state management
terraform {
  backend "azurerm" {
    resource_group_name     = azurerm_resource_group.rg.name
    storage_account_name    = azurerm_storage_account.terraform.name
    container_name          = azurerm_storage_container.terraform_state.name
    key                     = "terraform.tfstate"
  }
}
```

Initialize Terraform:

```
terraform init
```

---

3. GCP Configuration: Using Google Cloud Storage (GCS) for Terraform State

Objective: Store the Terraform state file in Google Cloud Storage (GCS).

Steps:

- **Create a GCS bucket for state storage**

Terraform Configuration:

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

# Create GCS Bucket for state storage
resource "google_storage_bucket" "terraform_state" {
  name     = "your-terraform-state-bucket"
  location = "US"
  storage_class = "STANDARD"
}

# Backend configuration to use GCS for state management
terraform {
  backend "gcs" {
    bucket = google_storage_bucket.terraform_state.name
    prefix = "terraform/state"
  }
}
```

Initialize Terraform:

```
terraform init
```

---

4. Oracle Cloud Configuration: Using Oracle Cloud Object Storage for Terraform State

Objective: Store the Terraform state file in Oracle Cloud Object Storage.

Steps:

- **Create an Object Storage bucket in Oracle Cloud**

Terraform Configuration:

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

# Create a bucket in Oracle Cloud Object Storage
resource "oci_objectstorage_bucket" "terraform_state" {
  compartment_id = "your-compartment-id"
  name           = "terraform-state-bucket"
  storage_tier   = "Standard"
}

# Backend configuration to use Oracle Cloud Object Storage for state management
terraform {
  backend "oci" {
    tenancy_ocid      = "your-tenancy-id"
    user_ocid         = "your-user-id"
    fingerprint       = "your-fingerprint"
    private_key_path  = "your-private-key-path"
    region            = "us-phoenix-1"
    bucket            = oci_objectstorage_bucket.terraform_state.name
    namespace         = "your-namespace"
    prefix            = "terraform/state"
  }
}
```

Initialize Terraform:

```
terraform init
```

General Steps for All Providers

1. **Create Terraform Configuration File**: Define resources for each cloud provider like S3 bucket (AWS), Blob Storage (Azure), GCS bucket (GCP), or Oracle Cloud Object Storage.
2. **Configure Backend**: In the terraform { backend "..." } block, specify the configuration for the respective provider.
3. **Initialize Terraform**: Use terraform init to initialize the working directory, download provider plugins, and set up the backend.
4. **Run Terraform Commands**:
   - terraform plan: Preview changes to be applied.
   - terraform apply: Apply the changes to infrastructure.
   - terraform state: Manage Terraform state.

Project 3: Compliance-as-Code

Overview: Automate compliance enforcement (CIS, HIPAA) using tools like Sentinel or OPA in combination with Terraform. This will help enforce compliance across various cloud platforms such as AWS, Azure, GCP, and Oracle.

---

1. AWS Compliance (CIS, HIPAA) using Sentinel

Sentinel allows you to integrate policy as code into Terraform to enforce compliance checks.

AWS Example Terraform Configuration:

```
provider "aws" {

  region = "us-west-2"

}



resource "aws_s3_bucket" "example_bucket" {

  bucket = "my-compliant-bucket"
  acl    = "private"
}

resource "aws_security_group" "example_sg" {
  name_prefix = "compliant-sg-"
  description = "Compliant security group"
}

# Sentinel Policy Example (Basic compliance check for an S3 bucket)
resource "sentinel_policy" "aws_compliance_policy" {
  name    = "aws_compliance"
  content = <<EOL
import "tfplan"
main = rule {
  tfplan.resource_changes["aws_s3_bucket.example_bucket"].after.acl == "private"
}
EOL
}
```

This policy ensures that the ACL for an S3 bucket is private, which is part of common compliance policies like CIS AWS Foundations.

---

2. Azure Compliance (CIS, HIPAA) using OPA

For Azure, integrate OPA (Open Policy Agent) with Terraform to enforce compliance rules like those defined in the CIS Azure Foundations.

Azure Example Terraform Configuration:

```
provider "azurerm" {

  features {}

}


resource "azurerm_resource_group" "example_rg" {

  name     = "example-rg"

  location = "East US"

}


resource "azurerm_storage_account" "example_storage" {

  name                     = "examplestorageacct"

  resource_group_name      = azurerm_resource_group.example_rg.name

  location                 = azurerm_resource_group.example_rg.location

  account_tier             = "Standard"

  account_replication_type = "LRS"

}


# Example OPA Policy for Azure (ensure storage accounts are encrypted)

data "opa_policy" "azure_compliance" {

  policy = <<EOL

package terraform.azure.storage_account
violation[resource] {
  resource = input.resource_changes[_]
  resource.type == "azurerm_storage_account"
  resource.after.encryption != "enabled"
}
EOL
}
```

This OPA policy checks whether Azure storage accounts have encryption enabled, which is a key compliance rule.

---

3. GCP Compliance (CIS, HIPAA) using OPA

For Google Cloud Platform (GCP), use OPA to enforce compliance policies like ensuring that GCP storage buckets are encrypted and IAM roles are configured properly.

GCP Example Terraform Configuration:

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

resource "google_storage_bucket" "example_bucket" {
  name     = "example-bucket-gcp"
  location = "US"

  encryption {

    default_kms_key =
"projects/my-project/locations/global/keyRings/my-key-ring/cryptoKeys/my-key"

  }

}


# Example OPA Policy for GCP (Ensure storage buckets are encrypted)

data "opa_policy" "gcp_compliance" {

  policy = <<EOL

package terraform.gcp.storage_bucket

violation[resource] {

  resource = input.resource_changes[_]

  resource.type == "google_storage_bucket"

  resource.after.encryption.default_kms_key == ""

}

EOL

}
```

This OPA policy ensures that GCP storage buckets are encrypted with the specified KMS key.

---

4. Oracle Cloud Compliance using Sentinel

For Oracle Cloud, Sentinel can be used similarly to enforce compliance policies, such as ensuring instances are launched in a VCN with proper security configurations.

Oracle Example Terraform Configuration:

```
provider "oci" {
  tenancy_ocid     = "ocid1.tenancy.oc1..aaaaaaaa"
  user_ocid        = "ocid1.user.oc1..aaaaaaaa"
  fingerprint      = "fingerprint"
  private_key_path = "~/.oci/private_key.pem"
  region           = "us-phoenix-1"
}

resource "oci_core_virtual_network" "example_vcn" {
  compartment_id = "ocid1.compartment.oc1..aaaaaaaa"
  display_name   = "example-vcn"
  cidr_block     = "10.0.0.0/16"
}

resource "oci_core_instance" "example_instance" {
  compartment_id       = "ocid1.compartment.oc1..aaaaaaaa"
  availability_domain  = "Uocm:PHX-AD-1"
  shape                = "VM.Standard2.1"
  display_name         = "example-instance"
  subnet_id            = oci_core_subnet.example_subnet.id
}

# Sentinel Policy for Oracle (Check that instance is created in a VCN)
resource "sentinel_policy" "oracle_compliance_policy" {
  name    = "oracle_compliance"
  content = <<EOL
import "tfplan"
main = rule {
  tfplan.resource_changes["oci_core_instance.example_instance"].after.subnet_id != ""
}
EOL
}
```

This Sentinel policy ensures that an Oracle Cloud instance is created in a valid subnet of an existing Virtual Cloud Network (VCN).

Conclusion

These configurations demonstrate how to use **Sentinel** and **OPA** in combination with **Terraform** to enforce compliance across **AWS**, **Azure**, **GCP**, and **Oracle Cloud** platforms. Depending on the compliance standards you are following (like **CIS**, **HIPAA**, etc.), you can extend these policies to include other checks such as IAM roles, encryption, logging, and more.

Project 4. Terraform for Compliance Automation

Create a Terraform project to automate the configuration of compliance requirements (like GDPR, HIPAA) using cloud security tools and policies.

Creating a **Terraform project for compliance automation** involves automating the configuration of compliance requirements like **GDPR** and **HIPAA** using cloud security tools and policies across different cloud providers **AWS**, **Azure**, **GCP**, and **Oracle Cloud**. Below are step-by-step guides to help you set up compliance automation for each cloud provider.

General Overview for Terraform Compliance Automation

1. **Compliance Requirements** GDPR, HIPAA, etc., have specific security standards, such as data encryption, access control, and auditing.
2. **Security Tools & Policies** Each cloud provider offers tools (e.g., IAM, Key Management, Security Hub) and policies to help automate compliance.
3. **Terraform Modules** Terraform allows us to define these policies as code, ensuring compliance at scale.

---

1. AWS - Compliance Automation with Terraform

Steps

1. **Set up AWS credentials**
   - Ensure AWS CLI is installed and configured with the appropriate credentials.
   - aws configure to set up your AWS credentials.
2. **Create Terraform files**
   - **main.tf** This will define the required AWS resources for compliance.
3. **Define IAM Policies for Compliance**
   - Use IAM roles, policies, and S3 bucket configurations to enforce GDPR or HIPAA requirements (e.g., encryption at rest).
   - Example Terraform configuration for GDPR compliance on S3 (enabling server-side encryption with AWS KMS)

```
resource "aws_s3_bucket" "compliance_bucket" {

  bucket = "compliance-data-bucket"

  acl    = "private"
  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "awskms"
        kms_master_key_id = "alias/aws/s3"
      }
    }
  }
}
```

4. **Set up CloudTrail for Auditing**
   - Configure AWS CloudTrail for logging compliance events.

```
resource "aws_cloudtrail" "compliance_trail" {
  name                          = "compliance-trail"
  s3_bucket_name                = "cloudtrail-logs-bucket"
  is_multi_region_trail         = true
  include_global_service_events = true
}
```

5. **Security Hub for Compliance Standards**
   - Enable AWS Security Hub to continuously check and enforce compliance standards like HIPAA.

```
resource "aws_securityhub_account" "example" {}
resource "aws_securityhub_standards_subscription" "hipaa" {
  standards_arn = "arnawssecurityhubus-west-2standards/hipaa"

}
```

6. **Apply the Configuration**
   - Run terraform init to initialize the configuration and terraform apply to deploy the resources.

---

2. Azure - Compliance Automation with Terraform

Steps

1. **Set up Azure credentials**
   - Configure the Azure CLI using az login.
2. **Create Terraform files**
   - **main.tf** This will define the required Azure resources for compliance.
3. **Define Azure Policies for Compliance**
   - Use Azure policies to enforce GDPR or HIPAA compliance. For instance, ensuring all storage accounts are encrypted.

```
resource "azurerm_storage_account" "compliance_storage" {

  name                     = "compstorageacct"

  resource_group_name      = azurerm_resource_group.rg.name

  location                 = azurerm_resource_group.rg.location

  account_tier             = "Standard"

  account_replication_type = "LRS"

  enable_https_traffic_only = true

  customer_managed_key_enabled = true

}
```

4. **Set up Azure Monitor for Auditing**
   - Use Azure Monitor to ensure audit logging for compliance.

```
resource "azurerm_monitor_diagnostic_setting" "compliance_diagnostics" {
  name                       = "compliance-logs"
  target_resource_id         = azurerm_storage_account.compliance_storage.id
  log_analytics_workspace_id = azurerm_log_analytics_workspace.example.id
  metric {
    category = "AllMetrics"
  }
}
```

5. **Set up Azure Security Center**
   - Enable Azure Security Center policies for compliance automation.

```
resource "azurerm_security_center_subscription_pricing" "example" {
  tier                = "Standard"
  subscription_id     = data.azurerm_client_config.example.subscription_id
  resource_group_name = azurerm_resource_group.example.name
}
```

6. **Apply the Configuration**
   - Run terraform init and terraform apply to deploy the configuration.

3. GCP - Compliance Automation with Terraform

Steps

1. **Set up GCP credentials**
   - Authenticate using the Google Cloud SDK (gcloud auth login).
2. **Create Terraform files**
   - **main.tf** This will define the required GCP resources for compliance.
3. **Define IAM Policies for Compliance**
   - Configure IAM roles to ensure proper access control for HIPAA/GDPR compliance.

```
resource "google_project_iam_member" "compliance_iam" {

  project = "my-project"

  role    = "roles/viewer"

  member  = "usercompliance@example.com"

}
```

4. **Set up Cloud Logging for Auditing**
   - Enable Cloud Audit Logs to maintain logs for compliance.

```
resource "google_logging_project_sink" "compliance_logs" {

  name        = "compliance-logs"

  project     = "my-project"

  destination = "storage.googleapis.com/compliance-logs-bucket"

  filter      = "logName('cloudaudit.googleapis.com')"

}
```

5. **Set up Cloud Security Command Center**
    - Enable Security Command Center for compliance auditing.

```
resource "google_security_center_settings" "example" {

  org_id = "your-org-id"

  enable = true

}
```

6. **Apply the Configuration**
    - Run terraform init and terraform apply to deploy the configuration.

---

4. Oracle Cloud - Compliance Automation with Terraform

Steps

1. **Set up Oracle Cloud credentials**
    - Configure the Oracle Cloud SDK (oci setup config).
2. **Create Terraform files**
    - **main.tf** This will define the required Oracle Cloud resources for compliance.
3. **Define IAM Policies for Compliance**
    - Use Oracle IAM to define policies for compliance, like limiting access based on roles.

```
resource "oci_identity_policy" "compliance_policy" {

  compartment_id = oci_identity_compartment.example.id

  name           = "compliance-policy"

  description    = "Enforces compliance policies"

  statements     = [

    "Allow group compliance-group to manage all-resources in compartment example"

  ]

}
```

4. **Set up Cloud Guard for Compliance Auditing**
   - Use Oracle Cloud Guard to continuously monitor for compliance violations.

```
resource "oci_cloud_guard_target" "compliance_target" {

  name           = "compliance-target"

  compartment_id   = oci_identity_compartment.example.id

  resource_type    = "all"

}
```

5. **Set up Logging for Auditing**
   - Enable logging to ensure compliance for auditing purposes.

```
resource "oci_logging_log_group" "compliance_log_group" {

  compartment_id = oci_identity_compartment.example.id

  name         = "compliance-log-group"

}
```

6. **Apply the Configuration**
   - Run terraform init and terraform apply to deploy the configuration.

Final Steps

1. **Check Compliance** After deploying resources, check for compliance using cloud-specific tools like AWS Security Hub, Azure Security Center, GCP Security Command Center, or Oracle Cloud Guard.
2. **Test Logging & Monitoring** Ensure logging and monitoring are set up to capture and alert on any compliance issues.
3. **Maintain & Update** Periodically update Terraform configurations to stay compliant with new policies and regulations.

This step-by-step guide should help you automate the configuration of compliance requirements (GDPR, HIPAA) using Terraform across **AWS**, **Azure**, **GCP**, and **Oracle Cloud**. You can modify these configurations to suit specific needs or compliance standards for your organization.

Project 5. Cloud-native API Security with Terraform

Set up infrastructure for API security by provisioning API gateways, firewalls, WAF (Web Application Firewall), and security policies using Terraform.

To set up **Cloud-native API Security** using **Terraform**, the project involves provisioning the necessary API security infrastructure across various cloud platforms such as AWS, Azure, GCP, and Oracle Cloud. Below is a step-by-step guide for each platform

1. AWS API Security with API Gateway, WAF, and Security Groups

Step 1 Install Terraform

Ensure Terraform is installed on your system. If it's not, install it from the Terraform website.

Step 2 Configure AWS Provider

```
provider "aws" {

  region = "us-west-2"

}
```

Step 3 Set Up API Gateway

Create an API Gateway that will act as the entry point for your APIs.

```
resource "aws_api_gateway_rest_api" "my_api" {

  name        = "my-api"

  description = "My API"

}
```

```
resource "aws_api_gateway_resource" "my_resource" {
  rest_api_id = aws_api_gateway_rest_api.my_api.id
  parent_id   = aws_api_gateway_rest_api.my_api.root_resource_id
  path_part   = "v1"
}
```

Step 4 Configure Web Application Firewall (WAF)

Provision a WAF to protect the API from attacks.

```
resource "aws_wafv2_web_acl" "my_waf" {
  name        = "my-waf"
  description = "My WAF for API Security"
  scope       = "REGIONAL" # Use "CLOUDFRONT" for CloudFront distributions
  default_action {
    allow {}
  }

  rules {
    name     = "IPAllowList"
    priority = 1
    action {
      allow {}
    }
    statement {
      ip_set_reference_statement {
        arn = aws_wafv2_ip_set.my_ip_set.arn
      }
    }
  }
}

resource "aws_wafv2_ip_set" "my_ip_set" {
  name   = "allow-ip-set"
  scope  = "REGIONAL"
  addresses = ["192.168.0.0/24"]
}
```

Step 5 Set Up Security Groups for API Gateway

Use AWS security groups to control inbound and outbound traffic.

```
resource "aws_security_group" "api_security_group" {
  name        = "api-security-group"
  description = "Allow access to API Gateway"
}

resource "aws_security_group_rule" "allow_inbound_http" {
  type        = "ingress"
  from_port   = 80
  to_port     = 80
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
  security_group_id = aws_security_group.api_security_group.id
}
```

Step 6 Apply the Terraform Configuration

Run the following commands to apply your Terraform configuration.

```
terraform init
terraform apply
```

---

2. Azure API Security with API Management, WAF, and NSG (Network Security Group)

Step 1 Install Terraform

Make sure Terraform is installed. If not, download it from Terraform's website.

Step 2 Configure Azure Provider

Add the Azure provider to your Terraform configuration.

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

Step 3 Set Up API Management

Provision an API Management service to manage your APIs securely.

```
resource "azurerm_api_management" "my_api" {
  name                = "my-api-management"
  location            = "East US"
  resource_group_name = "myResourceGroup"
  organization_name   = "MyOrg"
  sku_name            = "Consumption_1"
}

resource "azurerm_api_management_api" "my_api" {
  name                = "my-api"
  api_management_name = azurerm_api_management.my_api.name
  revision            = "1"
  display_name        = "My API"
  path                = "v1"
}
```

Step 4 Configure Web Application Firewall (WAF)

Provision WAF for API protection.

```
resource "azurerm_application_gateway" "my_waf" {
  name                = "my-waf"
  location            = "East US"
  resource_group_name = "myResourceGroup"
  frontend_ip_configuration {
    name                 = "my-frontend-ip"
    public_ip_address_id = azurerm_public_ip.my_public_ip.id
  }
  gateway_ip_configuration {
    name      = "my-gateway-ip"
    subnet_id = azurerm_subnet.my_subnet.id
  }

  waf_configuration {
    enabled = true
    firewall_mode = "Detection"
    rule_set_type = "OWASP"
    rule_set_version = "3.2"
  }
}
```

Step 5 Network Security Group (NSG)

Set up an NSG to control traffic to the API.

```
resource "azurerm_network_security_group" "api_security_group" {
  name                = "my-api-security-group"
  location            = "East US"
  resource_group_name = "myResourceGroup"
}
```

```
resource "azurerm_network_security_rule" "allow_inbound_http" {
  name                       = "allow-inbound-http"
  priority                   = 100
  direction                  = "Inbound"
  access                     = "Allow"
  protocol                   = "Tcp"
  source_port_range          = "*"
  destination_port_range     = "80"
  source_address_prefix      = "*"
  destination_address_prefix  = "*"
  network_security_group_name = azurerm_network_security_group.api_security_group.name
}
```

Step 6 Apply the Terraform Configuration

Run the following commands to deploy your configuration

```
terraform init
terraform apply
```

---

3. GCP API Security with API Gateway, Cloud Armor, and Firewall Rules

Step 1 Install Terraform

Download and install Terraform from Terraform’s website.

Step 2 Configure GCP Provider

Configure the GCP provider in Terraform.

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

Step 3 Set Up API Gateway

Create an API Gateway for managing your APIs.

```
resource "google_api_gateway_api" "my_api" {
  api_id = "my-api"
}

resource "google_api_gateway_api_config" "my_api_config" {
  api      = google_api_gateway_api.my_api.api_id
  config_id = "v1"
  openapi_documents {
    document {
      path = "openapi_spec.yaml"
    }
  }
}
resource "google_api_gateway_gateway" "my_gateway" {
  api      = google_api_gateway_api.my_api.api_id
  gateway_id = "my-gateway"
  location = "us-central1"
}
```

Step 4 Cloud Armor (WAF)

Provision Cloud Armor for API security.

```
resource "google_compute_security_policy" "my_waf" {
  name        = "my-waf"
  description = "WAF for API"
  rule {
    action = "allow"
    match {
      versioned_expr = "SRC_IPS_V1"
      config {
        src_ip_ranges = ["192.168.0.0/24"]
      }
    }
  }
}
```

Step 5 Firewall Rules

Create firewall rules to protect the API.

```
resource "google_compute_firewall" "api_firewall" {
  name    = "api-firewall"
  network = "default"
  allow {
    protocol = "tcp"
    ports    = ["80"]
  }
  source_ranges = ["0.0.0.0/0"]
}
```

Step 6 Apply the Terraform Configuration

Run these commands to apply the configuration

```
terraform init
terraform apply
```

---

4. Oracle Cloud API Security with API Gateway, WAF, and Security Lists

Step 1 Install Terraform

Install Terraform from Terraform’s website.

Step 2 Configure Oracle Cloud Provider

Set up the Oracle Cloud provider.

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

Step 3 Set Up API Gateway

Create an API Gateway.

```
resource "oci_apigateway_gateway" "my_api_gateway" {
  compartment_id = "your-compartment-id"
  display_name   = "my-api-gateway"
  endpoint_type  = "PUBLIC"
  subnet_id      = "your-subnet-id"
}
```

Step 4 Configure Web Application Firewall (WAF)

```
resource "oci_web_application_firewall_waf" "my_waf" {
  compartment_id = "your-compartment-id"
  display_name   = "my-waf"
  firewall_policy_id = "your-firewall-policy-id"
}
```

Step 5 Security Lists

Create security lists to control inbound traffic.

```
resource "oci_core_security_list" "api_security_list" {
  compartment_id = "your-compartment-id"
  display_name   = "api-security-list"
  ingress_security_rules {
    protocol = "6"
    source   = "0.0.0.0/0"
    tcp_options {
      destination_port_range {
        min = 80
        max = 80
      }
    }
  }
}
```

Step 6 Apply the Terraform Configuration

Execute these commands

```
terraform init
terraform apply
```

By following these step-by-step guides, you'll be able to provision API security infrastructure using Terraform across AWS, Azure, GCP, and Oracle Cloud, which includes setting up API Gateways, Web Application Firewalls, and configuring security groups, NSGs, and firewall rules for each platform.

Project 6. Cloud Security Best Practices

Automate the configuration of security-related resources, such as IAM roles, security policies, encryption, and monitoring for a cloud infrastructure project.

General Approach

1. **Install Terraform** Ensure Terraform is installed on your machine.
   - Terraform Installation Guide
2. **Set Up Cloud Provider Credentials**
   - AWS Configure AWS CLI using aws configure.
   - Azure Set up az CLI and log in using az login.
   - GCP Use gcloud auth login to authenticate.
   - Oracle Set up your Oracle Cloud credentials via oci CLI or configuration file.

---

1. AWS Cloud Security with Terraform

1. **Create IAM Roles**
   - Define IAM roles with permissions for security.

```
resource "aws_iam_role" "security_role" {

  name = "security-role"

  assume_role_policy = jsonencode({

    Version = "2012-10-17"

    Statement = [

      {

        Action = "stsAssumeRole"

        Effect = "Allow"
        Principal = {
          Service = "ec2.amazonaws.com"
        }
      }
    ]
  })
}
```

2. **Attach Security Policies**
   - Attach predefined AWS security policies.

```
resource "aws_iam_role_policy_attachment" "role_policy" {
  role       = aws_iam_role.security_role.name
  policy_arn = "arnawsiamawspolicy/AdministratorAccess"
}
```

3. **Encryption**
   - Enable encryption for S3 bucket or EBS volumes.

```
resource "aws_s3_bucket" "secure_bucket" {
  bucket = "secure-bucket-example"
  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "AES256"
      }
    }
  }
}
```

4. **Monitoring with CloudWatch**
   - Set up CloudWatch Alarms for monitoring.

```
resource "aws_cloudwatch_metric_alarm" "cpu_alarm" {
  alarm_name          = "high-cpu-alarm"
  comparison_operator = "GreaterThanThreshold"
  evaluation_periods  = "1"
  metric_name         = "CPUUtilization"
  namespace           = "AWS/EC2"
  period              = "60"
  statistic           = "Average"
  threshold           = "80"
  alarm_actions       = [aws_sns_topic.example.arn]
}
```

---

2. Azure Cloud Security with Terraform

1. **Create Managed Identity**
   - Define a managed identity for Azure resources.

```
resource "azurerm_user_assigned_identity" "example" {

  name                = "example-managed-identity"

  resource_group_name = azurerm_resource_group.example.name

  location            = azurerm_resource_group.example.location

}
```

2. **Define Role-Based Access Control (RBAC)**
   - Set up role assignments for secure access control.

```
resource "azurerm_role_assignment" "example" {

  principal_id   = azurerm_user_assigned_identity.example.principal_id

  role_definition_name = "Contributor"

  scope          = azurerm_resource_group.example.id

}
```

3. **Encryption**
   - Enable encryption for Azure Storage Accounts.

```
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"

  encryption {
    key_source = "Microsoft.Storage"

  }

}
```

4. **Monitoring with Azure Monitor**
    - Set up monitoring and alerts.

```
resource "azurerm_monitor_activity_log_alert" "example" {

  name                = "example-activity-log-alert"

  resource_group_name = azurerm_resource_group.example.name

  scopes              = [azurerm_resource_group.example.id]

  criteria {

    category = "Administrative"

    operation_name = "Delete"

  }

  action {

    action_group_id = azurerm_monitor_action_group.example.id

  }

}
```

---

3. GCP Cloud Security with Terraform

1. **Create IAM Roles**
    - Define custom IAM roles with secure permissions.

```
resource "google_project_iam_custom_role" "example" {
  role_id     = "custom-role"

  title       = "Custom Role"

  description = "Custom role for security"

  permissions = ["compute.instances.start", "compute.instances.stop"]

  project     = "your-project-id"

}
```

2. **Assign IAM Roles**
    - Assign the IAM role to users or service accounts.

```
resource "google_project_iam_binding" "example" {

  project = "your-project-id"

  role    = "roles/editor"

  members = [

    "userexample-user@gmail.com"

  ]

}
```

3. **Encryption**
    - Enable encryption for Google Cloud Storage buckets.

```
resource "google_storage_bucket" "example" {

  name     = "secure-bucket-gcp"

  location = "US"

  encryption {
    default_kms_key_name =
"projects/your-project-id/locations/global/keyRings/my-key-ring/cryptoKeys/my-key"

  }

}
```

4. **Monitoring with Google Cloud Monitoring**
   - Create an alerting policy in Cloud Monitoring.

```
resource "google_monitoring_alert_policy" "example" {

  display_name = "High CPU Alert"

  notification_channels = [google_monitoring_notification_channel.email.id]

  conditions {

    display_name = "High CPU Utilization"

    condition_threshold {

      comparison = "COMPARISON_GT"

      threshold_value = 80

      filter = "metric.type=\"compute.googleapis.com/instance/disk/write_bytes_count\" AND
resource.type=\"gce_instance\""

    }

  }

}
```

---

4. Oracle Cloud Security with Terraform

1. **Create IAM Policy**
   - Define policies for secure access.

```
resource "oci_identity_policy" "example" {
  name             = "secure-policy"

  compartment_id    = "your-compartment-id"

  statements        = ["Allow group Administrators to manage all-resources in compartment
Your-Compartment"]

}
```

2. **Create Encryption Keys**
   - Define encryption keys for secure storage.

```
resource "oci_kms_key" "example" {

  compartment_id = "your-compartment-id"

  display_name   = "example-encryption-key"

  key_shape {

    algorithm = "AES"

    length    = 256

  }

}
```

3. **Monitoring with Oracle Cloud Monitoring**
   - Set up alerts and alarms.

```
resource "oci_monitoring_alarm" "example" {

  compartment_id = "your-compartment-id"

  name           = "high-cpu-alarm"

  metric {
    namespace = "oci_computeagent"
    name      = "CPUUtilization"
    dimension = {
      "resourceId" = "your-instance-id"
    }
  }
  threshold {
    comparison = "GREATER_THAN"
    value      = 80
  }
}
```

Conclusion

- For **AWS**, **Azure**, **GCP**, and **Oracle Cloud**, the Terraform scripts automate the creation of IAM roles, security policies, encryption mechanisms, and monitoring setup.
- Make sure to replace placeholders (e.g., your-project-id, your-compartment-id) with actual values.
- After setting up Terraform configurations, use terraform init, terraform plan, and terraform apply to provision resources.

8. Cost Optimization & Monitoring

Project 1. Infrastructure Cost Optimization

Use Terraform to manage spot or reserved instances, tagging, and cost analysis tools.

Here is a basic overview of Terraform configurations for managing infrastructure cost optimization on AWS, Azure, GCP, and Oracle Cloud. These configurations focus on using **spot or reserved instances**, **tagging resources**, and leveraging **cost analysis tools**.

AWS Using Spot Instances, Reserved Instances, and Tagging

```
# AWS Provider Configuration

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

# Reserved EC2 Instances

resource "aws_instance" "reserved" {
  instance_type = "t2.micro"
  ami           = "ami-0c55b159cbfafe1f0" # example AMI
  count         = 1
  tags = {
    Name = "Reserved Instance"
    Environment = "Production"
  }


  lifecycle {
    create_before_destroy = true
  }
}

# Spot EC2 Instances

resource "aws_instance" "spot" {
  instance_type = "t2.micro"
  ami           = "ami-0c55b159cbfafe1f0" # example AMI
  spot_price    = "0.03"  # Spot price for instance
  count         = 2
  tags = {
    Name = "Spot Instance"
    Environment = "Development"
  }
}
# Tagging for Cost Allocation
resource "aws_resourcegroupstaggingapi_tag" "tag_cost_allocation" {
  resource_arn = aws_instance.reserved.arn
  key          = "CostCenter"
  value        = "DevOps"
}
```

Azure Using Reserved Virtual Machines and Tagging

```
# Azure Provider Configuration
provider "azurerm" {
  features {}
}
```

```
# Reserved Virtual Machine
resource "azurerm_virtual_machine" "reserved" {
  name                  = "reservedVM"
  location              = "East US"
  resource_group_name   = "myResourceGroup"
  size                  = "Standard_B1ms"
  network_interface_ids = [azurerm_network_interface.nic.id]
  vm_os_type            = "Linux"
  admin_username        = "adminuser"
  admin_password        = "adminpassword"
  tags = {
    CostCenter = "CloudOps"
  }
}

# Spot Virtual Machine
resource "azurerm_virtual_machine" "spot" {
  name                  = "spotVM"
  location              = "East US"
  resource_group_name   = "myResourceGroup"
  size                  = "Standard_B1ms"
  network_interface_ids = [azurerm_network_interface.nic.id]
  vm_os_type            = "Linux"
  admin_username       = "adminuser"
  admin_password       = "adminpassword"
  priority             = "Spot"  # Spot VM for cost-saving
  tags = {
    CostCenter = "Development"
  }
}
```

Google Cloud Platform Using Preemptible VMs and Tagging

# GCP Provider Configuration

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

# Preemptible Instance

```
resource "google_compute_instance" "preemptible" {
  name         = "preemptible-vm"
  machine_type = "n1-standard-1"
  zone         = "us-central1-a"
  preemptible  = true
  tags = ["cost-optimization"]

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

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

# Tagging for cost allocation
resource "google_compute_instance" "tagged_vm" {
  name         = "tagged-vm"
  machine_type = "n1-standard-1"
  zone         = "us-central1-a"

  tags = ["prod", "envdev"]
```

```
}
```

Oracle Cloud Using Always Free and Cost Management Tagging

```
# Oracle Cloud Provider Configuration
provider "oci" {
  region = "us-phoenix-1"
}

# Always Free Instance
resource "oci_core_instance" "free_instance" {
  availability_domain = data.oci_identity_availability_domains.ads.names[0]
  compartment_id      = var.compartment_id
  shape               = "VM.Standard.E2.1.Micro"  # Always free eligible shape
  display_name        = "FreeInstance"

  create_vnic_details {
    subnet_id = var.subnet_id
    assign_public_ip = true
  }

  metadata = {
    ssh_authorized_keys = var.ssh_public_key

  }



  tags = {

    CostCenter = "FreeTier"

  }

}

# Tagging for Cost Allocation

resource "oci_identity_tag" "cost_center_tag" {

  compartment_id = var.compartment_id

  name           = "CostCenter"

}
```

Key Points to Consider

Spot and Reserved Instances These configurations help optimize costs by utilizing **spot instances** for non-critical workloads and **reserved instances** for long-term stable workloads. Spot instances are cheaper but can be terminated, while reserved instances offer significant savings for long-term usage.

Tagging Proper tagging helps in cost allocation and management. For example, the CostCenter tag is useful for tracking usage across different teams and projects.

Cost Analysis Tools To further analyze costs, use the built-in cost management tools in each cloud provider like **AWS Cost Explorer**, **Azure Cost Management**, **GCP Billing**, and **Oracle Cloud Cost Management**.

Project 2. Cost Optimization

Use Terraform to create and manage auto-scaling groups, right-size instances, and use spot instances for cost-efficient infrastructure.

Step 1 Install Terraform

Ensure you have Terraform installed on your system. If it's not already installed, follow these steps

1. Download Terraform from Terraform's official website.
2. Follow the installation instructions for your platform (Linux, macOS, or Windows).

Step 2 Set up Your AWS Account and Credentials

For this project, you'll need an AWS account. Set up your credentials for Terraform to access your AWS environment

1. **Create an IAM User** in AWS with AdministratorAccess (or a suitable permission for Terraform to create resources).

Configure AWS CLI (if you haven't already) using the following command

aws configure

Enter your AWS Access Key ID, Secret Access Key, region, and output format.

Verify your credentials are set up properly by running

aws sts get-caller-identity

Step 3 Initialize Your Terraform Project

Create a new directory for your Terraform project

mkdir terraform-cost-optimization

cd terraform-cost-optimization

Inside this directory, create the following files

1. **main.tf** - to define your resources.
2. **variables.tf** - to define any variables.
3. **outputs.tf** - to output important details about your resources.

Step 4 Define Your AWS Provider in main.tf

Start by defining the AWS provider in your main.tf

```
provider "aws" {

  region = "us-east-1"  # Change to your desired region

}
```

Step 5 Create an EC2 Instance

You can create a base EC2 instance to use as a launch template for your auto-scaling group. Here's a simple EC2 instance definition

```
resource "aws_instance" "example" {

  ami           = "ami-0c55b159cbfafe1f0"  # Replace with a valid AMI ID in your region

  instance_type = "t2.micro"  # Change to your preferred instance type


  tags = {

    Name = "ExampleInstance"

  }
}
```

Step 6 Set Up Auto-Scaling Group

Create an Auto Scaling Group (ASG) that will automatically scale your infrastructure. Define a launch configuration for your instances

```
resource "aws_launch_configuration" "example" {
  name          = "example-launch-config"
  image_id      = "ami-0c55b159cbfafe1f0"  # Replace with your AMI ID
  instance_type = "t2.micro"  # Adjust the instance type as needed

  lifecycle {
    create_before_destroy = true
  }
}
resource "aws_autoscaling_group" "example" {
  desired_capacity     = 2
  min_size             = 1
  max_size             = 3
  vpc_zone_identifier  = ["subnet-abc12345"]  # Replace with your subnet ID
  launch_configuration = aws_launch_configuration.example.id
}
```

This setup will create an Auto Scaling Group with a desired capacity of 2, and it will automatically scale between 1 and 3 instances depending on the load.

Step 7 Right-size Instances (optional)

}

For right-sizing, you can either manually specify a more cost-effective instance type or use Terraform data sources to get metrics on instance performance. In this example, you'll manually choose an instance type that's more cost-effective.

To optimize cost, you can use t3.micro or t3.small instance types, which offer better cost efficiency for certain workloads.

Step 8 Use Spot Instances for Cost Efficiency

To use Spot Instances in the Auto Scaling Group, modify the aws_autoscaling_group resource to use a Spot price for instances

```
resource "aws_autoscaling_group" "example" {

  desired_capacity     = 2

  min_size             = 1

  max_size             = 3

  vpc_zone_identifier  = ["subnet-abc12345"]  # Replace with your subnet ID

  launch_configuration = aws_launch_configuration.example.id


  mixed_instances_policy {

    instances_distribution {

      spot_price = "0.03"  # Set the spot price that you're willing to pay per hour

    }

    launch_template {

      launch_template_name = aws_launch_configuration.example.name

      version              = "$Latest"

    }

  }

}
```

Spot instances are a great way to save on infrastructure costs, as they allow you to take advantage of unused EC2 capacity.

Step 9 Add Auto-Scaling Policies

You can add policies for auto-scaling to ensure that the number of instances adjusts based on the load. Example for scaling out when CPU usage is high

```
resource "aws_autoscaling_policy" "scale_up" {

  name                   = "scale-up"

  scaling_adjustment     = 1

  adjustment_type        = "ChangeInCapacity"

  cooldown               = 300

  autoscaling_group_name   = aws_autoscaling_group.example.name

}


resource "aws_cloudwatch_metric_alarm" "cpu_high" {

  alarm_name          = "high-cpu"

  comparison_operator = "GreaterThanThreshold"

  evaluation_periods  = 2

  metric_name         = "CPUUtilization"

  namespace           = "AWS/EC2"

  period              = 60

  statistic           = "Average"

  threshold           = 75

  alarm_actions       = [aws_autoscaling_policy.scale_up.arn]
  dimensions = {
    AutoScalingGroupName = aws_autoscaling_group.example.name
  }
}
```

Step 10 Terraform Initialization and Apply

Initialize the Terraform project to install required provider plugins
terraform init

Validate the Terraform configuration
terraform validate

Plan the infrastructure changes to review what Terraform will do
terraform plan

Apply the configuration to create the resources
terraform apply

Confirm the prompt by typing yes.

Step 11 Monitor and Optimize Costs

After applying your changes, you can monitor your infrastructure using the AWS Management Console, CloudWatch, or by setting up further cost management services like AWS Cost Explorer.

Project 3. Cloud Monitoring Setup

Use Terraform to deploy cloud monitoring tools like Prometheus, Grafana, or CloudWatch on your infrastructure.

1. AWS Setting up Prometheus and Grafana

Prerequisites

- AWS account
- Terraform installed
- IAM user with permissions to deploy infrastructure

Steps

Configure AWS Provider in Terraform

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

Create Security Group This will allow communication between Prometheus, Grafana, and other services.

```
resource "aws_security_group" "monitoring_sg" {
  name        = "monitoring_sg"
  description = "Allow inbound communication for Prometheus and Grafana"
  ingress {
    from_port   = 3000
    to_port     = 3000
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ingress {
    from_port   = 9090
    to_port     = 9090
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Deploy Prometheus (EC2 instance)

```
resource "aws_instance" "prometheus" {

  ami           = "ami-xxxxxxxx"  # Replace with a valid Ubuntu AMI

  instance_type = "t2.micro"

  security_groups = [aws_security_group.monitoring_sg.name]

  user_data = <<-EOF

    #!/bin/

    wget
https//github.com/prometheus/prometheus/releases/download/v2.34.0/prometheus-2.34.0.linux-a
md64.tar.gz

    tar -xvf prometheus-2.34.0.linux-amd64.tar.gz

    cd prometheus-2.34.0.linux-amd64

    ./prometheus --config.file=prometheus.yml

  EOF

}
```

Deploy Grafana (EC2 instance)

```
resource "aws_instance" "grafana" {

  ami           = "ami-xxxxxxxx"  # Replace with a valid Ubuntu AMI

  instance_type = "t2.micro"

  security_groups = [aws_security_group.monitoring_sg.name]

  user_data = <<-EOF

    #!/bin/
    sudo apt-get install -y software-properties-common

    sudo add-apt-repository "deb https//packages.grafana.com/oss/deb stable main"

    sudo apt-get update

    sudo apt-get install grafana

    sudo systemctl start grafana-server

    sudo systemctl enable grafana-server

  EOF

}
```

Output Grafana URL

```
output "grafana_url" {

  value = aws_instance.grafana.public_ip

}
```

Run Terraform Initialize, plan, and apply the configuration.

```
terraform init

terraform plan

terraform apply
```

---

2. Azure Setting up Prometheus and Grafana

Prerequisites

- Azure account

- Terraform installed
- Azure CLI configured

Steps

Configure Azure Provider in Terraform

```
provider "azurerm" {

  features {}

}
```

Create a Virtual Network

```
resource "azurerm_virtual_network" "vnet" {

  name                = "monitoring-vnet"

  address_space       = ["10.0.0.0/16"]

  location            = "East US"

  resource_group_name = azurerm_resource_group.rg.name

}
```

Create Prometheus VM (Ubuntu-based)

```
resource "azurerm_linux_virtual_machine" "prometheus" {

  name                = "prometheus-vm"

  resource_group_name = azurerm_resource_group.rg.name

  location            = "East US"

  size                = "Standard_B1s"

  admin_username      = "adminuser"

  admin_password      = "password1234"
  network_interface_ids = [azurerm_network_interface.nic.id]


  custom_data = <<-EOF
    #!/bin/
    wget
https//github.com/prometheus/prometheus/releases/download/v2.34.0/prometheus-2.34.0.linux-a
md64.tar.gz
    tar -xvf prometheus-2.34.0.linux-amd64.tar.gz
    cd prometheus-2.34.0.linux-amd64
    ./prometheus --config.file=prometheus.yml
  EOF
}
```

Create Grafana VM (Ubuntu-based)

```
resource "azurerm_linux_virtual_machine" "grafana" {
  name                = "grafana-vm"
  resource_group_name = azurerm_resource_group.rg.name
  location            = "East US"
  size                = "Standard_B1s"
  admin_username      = "adminuser"
  admin_password      = "password1234"


  network_interface_ids = [azurerm_network_interface.nic.id]
  custom_data = <<-EOF

    #!/bin/

    sudo apt-get install -y software-properties-common

    sudo add-apt-repository "deb https//packages.grafana.com/oss/deb stable main"

    sudo apt-get update

    sudo apt-get install grafana

    sudo systemctl start grafana-server

    sudo systemctl enable grafana-server

  EOF

}
```

Run Terraform

```
terraform init

terraform plan

terraform apply
```

---

3. GCP Setting up Prometheus and Grafana

Prerequisites

- GCP account
- Terraform installed
- Google Cloud SDK configured

Steps

Configure GCP Provider in Terraform

```
provider "google" {

  project = "my-gcp-project"

  region  = "us-central1"

}
```

Create VM Instances for Prometheus and Grafana

```
resource "google_compute_instance" "prometheus" {

  name         = "prometheus-vm"

  machine_type = "f1-micro"

  zone         = "us-central1-a"

  tags         = ["http-server", "https-server"]


  boot_disk {

    initialize_params {

      image = "ubuntu-2004-focal-v20210927"

    }

  }


  network_interface {

    network = "default"

    access_config {

      // Allocate a public IP

    }
```

```
 }

 metadata_startup_script = <<-EOT
   #!/bin/
   wget
https//github.com/prometheus/prometheus/releases/download/v2.34.0/prometheus-2.34.0.linux-a
md64.tar.gz
   tar -xvf prometheus-2.34.0.linux-amd64.tar.gz
   cd prometheus-2.34.0.linux-amd64
   ./prometheus --config.file=prometheus.yml
 EOT
}

resource "google_compute_instance" "grafana" {
 name         = "grafana-vm"
 machine_type = "f1-micro"
 zone         = "us-central1-a"
 tags         = ["http-server", "https-server"]

 boot_disk {
   initialize_params {
     image = "ubuntu-2004-focal-v20210927"
   }
 }
```

```
  network_interface {

    network = "default"

    access_config {

      // Allocate a public IP

    }

  }


  metadata_startup_script = <<-EOT

    #!/bin/

    sudo apt-get install -y software-properties-common

    sudo add-apt-repository "deb https//packages.grafana.com/oss/deb stable main"

    sudo apt-get update

    sudo apt-get install grafana

    sudo systemctl start grafana-server

    sudo systemctl enable grafana-server

  EOT

}
```

Run Terraform

```
terraform init

terraform plan

terraform apply
```

4. Oracle Cloud Setting up Prometheus and Grafana

Prerequisites

- Oracle Cloud account
- Terraform installed
- OCI CLI configured

Steps

Configure OCI Provider in Terraform

```
provider "oci" {

  tenancy_ocid         = "<TENANCY_OCID>"

  user_ocid            = "<USER_OCID>"

  fingerprint          = "<FINGERPRINT>"

  private_key_path     = "~/.oci/oci_api_key.pem"

  region               = "us-phoenix-1"

}
```

Create Compute Instance for Prometheus

```
resource "oci_core_instance" "prometheus" {

  availability_domain = "UocmPHX-AD-1"

  compartment_id      = "<COMPARTMENT_OCID>"

  shape               = "VM.Standard2.1"

  display_name        = "Prometheus-Instance"



  create_vnic_details {

    subnet_id = "<SUBNET_OCID>"
  }

  source_details {
    source_type = "image"
    image_id    = "<IMAGE_OCID>"
  }

  metadata = {
    ssh_authorized_keys = "<SSH_PUBLIC_KEY>"
  }
}
```

Create Compute Instance for Grafana

```
resource "oci_core_instance" "grafana" {
  availability_domain = "UocmPHX-AD-1"
  compartment_id      = "<COMPARTMENT_OCID>"
  shape               = "VM.Standard2.1"
  display_name        = "Grafana-Instance"

  create_vnic_details {
    subnet_id = "<SUBNET_OCID>"
  }

  source_details {
    source_type = "image"

    image_id    = "<IMAGE_OCID>"

  }


  metadata = {

    ssh_authorized_keys = "<SSH_PUBLIC_KEY>"

  }

}
```

Run Terraform

```
terraform init

terraform plan

terraform apply
```

Project 4. Automated Monitoring and Alerts

Set up cloud-based monitoring (e.g., AWS CloudWatch, GCP Monitoring) using Terraform to monitor the health of infrastructure components and configure alerts.

AWS CloudWatch Monitoring and Alerts

1. **Prerequisites**
    - AWS account
    - Terraform installed
    - AWS CLI configured with necessary permissions

Create Terraform Configuration File
In your project directory, create a main.tf file.

```
provider "aws" {

  region = "us-west-2"  # Specify your AWS region

}


# Create CloudWatch Alarm for EC2 CPU Utilization

resource "aws_cloudwatch_metric_alarm" "cpu_alarm" {

  alarm_name                = "EC2-CPU-Utilization-Alarm"

  comparison_operator       = "GreaterThanThreshold"

  evaluation_periods        = 1

  metric_name               = "CPUUtilization"

  namespace                 = "AWS/EC2"

  period                    = 300

  statistic                 = "Average"

  threshold                 = 80


  dimensions = {

    InstanceId = "i-xxxxxxxxxxxxxxx"  # Replace with your EC2 instance ID

  }


  alarm_actions = ["arnawssnsus-west-2123456789012MySNSTopic"]  # Replace with SNS topic
ARN
```

```
  ok_actions    = ["arnawssnsus-west-2123456789012MySNSTopic"]  # Replace with SNS topic
ARN

}
```

# Create SNS Topic for Notification

```
resource "aws_sns_topic" "cloudwatch_topic" {

  name = "MySNSTopic"

}
```

# Create an SNS Subscription (email)

```
resource "aws_sns_topic_subscription" "email_subscription" {

  topic_arn = aws_sns_topic.cloudwatch_topic.arn

  protocol  = "email"

  endpoint  = "your-email@example.com"  # Replace with your email

}
```

Apply Terraform Configuration Run the following Terraform commands to deploy the resources.

```
terraform init

terraform apply
```

Verify

- Ensure the CloudWatch Alarm is created in the AWS console.
- Check for email alerts based on the configured thresholds.

Azure Monitor and Alerts

1. **Prerequisites**
   - Azure account
   - Terraform installed
   - Azure CLI configured with necessary permissions

Create Terraform Configuration File
Create a main.tf file with the following content

```
provider "azurerm" {

  features {}

}


# Create a Log Analytics Workspace

resource "azurerm_log_analytics_workspace" "example" {

  name                = "example-workspace"

  location            = "East US"

  resource_group_name = "example-resources"

  sku                 = "PerGB2018"

}


# Create an Azure Monitor Metric Alert

resource "azurerm_monitor_metric_alert" "cpu_alert" {

  name                = "cpu-alert"

  resource_group_name = "example-resources"
  target_resource_id  =
"/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.Co
mpute/virtualMachines/{vmName}"

  criteria {

    metric_name       = "Percentage CPU"

    aggregation       = "Average"

    operator          = "GreaterThan"

    threshold         = 80

    time_aggregation   = "Average"

  }


  action {

    action_group_id =
"/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/microsoft.insi
ghts/actionGroups/{actionGroupName}"

  }

}
```

Apply Terraform Configuration Run the following Terraform commands to deploy the resources.

```
terraform init

terraform apply
```

Verify

- Ensure the Azure Monitor Metric Alert and Log Analytics Workspace are created in the Azure portal.
- Check for email or webhook notifications triggered by the alert.

Google Cloud Monitoring (GCP) and Alerts

1. **Prerequisites**
    - GCP account
    - Terraform installed
    - GCP credentials configured

Create Terraform Configuration File
Create a main.tf file with the following content

```
provider "google" {

  project = "your-project-id"

  region  = "us-central1"

}


# Create an alerting policy in GCP Monitoring

resource "google_monitoring_alert_policy" "cpu_alert" {

  display_name = "CPU Utilization Alert"

  notification_channels = [

    "projects/your-project-id/notificationChannels/your-channel-id"

  ]
```

```
 conditions {
  display_name = "CPU Utilization Condition"
  condition_threshold {
   comparison = "COMPARISON_GT"
   threshold_value = 80
   aggregations {
    alignment_period   = "60s"
    per_series_aligner = "ALIGN_RATE"
   }
   metric {
    type = "compute.googleapis.com/instance/disk/write_bytes_count"
    labels {
     key   = "instance_id"
     value = "your-instance-id"
    }
   }
  }
 }
}
```

Apply Terraform Configuration Run the following Terraform commands to deploy the resources.

```
terraform init
terraform apply
```

Verify

- Ensure the alerting policy and notification channels are configured in GCP.
- Check for email notifications based on the alert thresholds.

Oracle Cloud Monitoring and Alerts

1. **Prerequisites**
   - Oracle Cloud account
   - Terraform installed
   - OCI CLI configured with necessary permissions

Create Terraform Configuration File
Create a main.tf file with the following content

```
provider "oci" {

  tenancy_ocid        = "ocid1.tenancy.oc1..aaaaaaaaxxxxx"

  user_ocid           = "ocid1.user.oc1..aaaaaaaaxxxxx"

  fingerprint         = "xxxxxxxxxx"

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

  region              = "us-phoenix-1"

}


# Create an Alarm in Oracle Cloud

resource "oci_monitoring_alarm" "cpu_alarm" {

  compartment_id = "ocid1.compartment.oc1..aaaaaaaaxxxxx"

  display_name   = "CPU Utilization Alert"
  severity      = "CRITICAL"

  metric_query {
    aggregation = "AVG"
    metric {
      namespace = "oci_computeagent"
      name      = "CpuUtilization"
      dimensions = {
        "resourceId" = "your-instance-id"
      }
    }
    threshold {
      comparison = "GT"
      value      = 80
    }
  }

  notification {
    method = "EMAIL"
    destination = "your-email@example.com"  # Replace with your email
  }
}
```

Apply Terraform Configuration Run the following Terraform commands to deploy the resources.

terraform init

terraform apply

2. **Verify**
   - Ensure the Oracle Cloud Monitoring Alarm is set up in the Oracle Console.
   - Check for email notifications based on the alert configuration.

Final Steps for All Platforms

- Make sure the necessary permissions and roles are set up (e.g., IAM roles for monitoring and alerting).
- Test the alerts by simulating load on your infrastructure (e.g., increasing CPU usage).
- Configure any additional actions based on your alert needs (e.g., webhook triggers, Lambda functions).

Project 5. Cost and Usage Monitoring with AWS Budgets

Create a Terraform setup for AWS Budgets, monitoring usage and setting up alerts for cloud spending thresholds to ensure cost management.

1. AWS Cost and Usage Monitoring with AWS Budgets

Objective Use Terraform to create AWS Budgets, monitor usage, and set up alerts for cloud spending thresholds.

Step 1 Set up AWS Credentials

Ensure you have AWS CLI configured with the correct credentials.

```
aws configure
```

Step 2 Install Terraform

Install Terraform on your machine if you haven't already.

Step 3 Create a Terraform Configuration File

Create a Terraform file (e.g., aws_budget.tf) to define the resources.

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

resource "aws_budgets_budget" "cost_budget" {
  name              = "MonthlyCostBudget"
  budget_type       = "COST"
  time_unit         = "MONTHLY"
  budget_limit {
    amount = "100"
    unit   = "USD"
  }

  cost_filters = {
    "LinkedAccount" = "123456789012"
  }

  time_period {
    start = "2024-11-01T000000Z"
    end   = "2025-11-01T000000Z"
  }

  notification {
    comparison_operator = "GREATER_THAN"
    threshold          = 80
    threshold_type     = "PERCENTAGE"
    subscriber {
      subscription_type = "EMAIL"
      address           = "your-email@example.com"
    }
  }

  notification {
    comparison_operator = "GREATER_THAN"
    threshold          = 100
    threshold_type     = "PERCENTAGE"
    subscriber {
      subscription_type = "EMAIL"
      address           = "your-email@example.com"
    }
  }
}
```

Step 4 Initialize Terraform

Run this command to initialize the Terraform configuration.

```
terraform init
```

Step 5 Apply the Terraform Configuration

Run this command to create the budget.

```
terraform apply
```

---

2. Azure Cost Management with Azure Budgets

Objective Use Terraform to create Azure Budgets for monitoring usage and setting cost thresholds.

Step 1 Set up Azure Credentials

Make sure you have the Azure CLI installed and configured.

az login

Step 2 Create a Terraform Configuration File for Azure

Create a new Terraform configuration (e.g., azure_budget.tf).

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

resource "azurerm_cost_management_budget" "monthly_budget" {
  name              = "monthly-budget"
  scope             = "/subscriptions/${var.subscription_id}"
  amount            = 100
  time_grain        = "Monthly"
  time_period_start  = "2024-11-01T000000Z"
  time_period_end    = "2025-11-01T000000Z"

  notification {
    enabled = true
    threshold = 80
    operator  = "GreaterThan"
    contact_emails = ["your-email@example.com"]
  }

  notification {
    enabled = true
    threshold = 100
    operator  = "GreaterThan"
    contact_emails = ["your-email@example.com"]
  }
}
```

Step 3 Initialize Terraform

Run this command to initialize the configuration.

```
terraform init
```

Step 4 Apply the Terraform Configuration

Run this command to apply the configuration.

```
terraform apply
```

---

3. GCP Cost Management with Budgets and Alerts

Objective Use Terraform to create GCP Budgets and set up alerts for cost management.

Step 1 Set up GCP Credentials

Make sure you have the gcloud CLI and Terraform provider configured.

```
gcloud auth login
```

Step 2 Create a Terraform Configuration File for GCP

Create a new file (e.g., gcp_budget.tf).

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

resource "google_billing_budget" "monthly_budget" {
  billing_account = "billingAccounts/your-billing-account-id"
  display_name    = "Monthly Budget"

  budget_filter {
    credit_types_treatment = "INCLUDE_ALL"
  }
  amount {
    specified_amount {
      currency_code = "USD"
      units         = 100
    }
  }

  threshold_rule {
    threshold_percent = 0.8
    spend_basis       = "CURRENT_SPEND"
    notification_channel = "projects/your-gcp-project-id/notifications/your-notification-id"
  }

  threshold_rule {
    threshold_percent = 1.0
    spend_basis       = "CURRENT_SPEND"
    notification_channel = "projects/your-gcp-project-id/notifications/your-notification-id"
  }
}
```

Step 3 Initialize Terraform

Run the initialization command for GCP.

```
terraform init
```

Step 4 Apply the Configuration

Apply the changes to create the budget.

```
terraform apply
```

---

4. Oracle Cloud Cost Management with Budgets

Objective Use Terraform to create Oracle Cloud Budgets for cost management.

Step 1 Set up Oracle Cloud Credentials

Make sure you have Oracle Cloud CLI configured.

```
oci setup config
```

Step 2 Create a Terraform Configuration File for Oracle

Create a new configuration file (oracle_budget.tf).

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

resource "oci_billing_budget" "monthly_budget" {
  compartment_id = "ocid1.compartment.oc1..your-compartment-id"
  budget_type    = "COST"
  amount         = 100
  time_unit      = "MONTHLY"

  notification {
    threshold        = 80
    comparison_type  = "GREATER_THAN"
    notification_type = "EMAIL"
    recipient_email  = "your-email@example.com"
  }

  notification {
    threshold        = 100
    comparison_type  = "GREATER_THAN"
    notification_type = "EMAIL"
    recipient_email  = "your-email@example.com"
  }
}
```

Step 3 Initialize Terraform

Initialize the Terraform configuration.

terraform init

Step 4 Apply the Terraform Configuration

Apply the configuration to create the budget.

terraform apply

Conclusion

By following the steps for each cloud provider (AWS, Azure, GCP, Oracle), you can set up cost and usage monitoring with budgets and alerts using Terraform. Each provider has its own syntax and requirements for configuring budgets, but the overall process of defining budgets, setting thresholds, and enabling alerts is similar.

9. Databases & Storage

Project 1. Database Setup and Configuration

AWS: Database Setup Project

Directory Structure:

```
aws-database-setup/

├── main.tf

├── variables.tf

├── outputs.tf
├── terraform.tfvars
```

Files:

main.tf

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

resource "aws_rds_instance" "db_instance" {
  allocated_storage    = 20
  engine               = "mysql"
  instance_class       = "db.t2.micro"
  name                 = "awsdb"
  username             = var.db_username
  password             = var.db_password
  parameter_group_name = "default.mysql8.0"
}
```

variables.tf

```
variable "region" {}

variable "db_username" {}

variable "db_password" {}
```

outputs.tf

```
output "db_endpoint" {
  value = aws_rds_instance.db_instance.endpoint
}
```

terraform.tfvars

```
region      = "us-east-1"
db_username = "admin"
db_password = "securepassword"
```

Azure: Database Setup Project

Directory Structure:

```
azure-database-setup/
├── main.tf
├── variables.tf
├── outputs.tf
├── terraform.tfvars
```

Files:

main.tf

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

resource "azurerm_mysql_flexible_server" "db_server" {
  name                = "azuredb"
  location            = var.location
  resource_group_name = var.resource_group_name
  admin_username      = var.db_username
  admin_password      = var.db_password
  version             = "8.0"
}

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

variables.tf

```
variable "location" {}
variable "resource_group_name" {}
variable "db_username" {}
variable "db_password" {}
```

outputs.tf

```
output "db_server_name" {
  value = azurerm_mysql_flexible_server.db_server.name
}
```

`terraform.tfvars`

```
location            = "East US"

resource_group_name = "AzureDBRG"

db_username         = "admin"

db_password         = "securepassword"
```

GCP: Database Setup Project

Directory Structure:

```
gcp-database-setup/

├── main.tf

├── variables.tf

├── outputs.tf

├── terraform.tfvars
```

Files:

`main.tf`

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

resource "google_sql_database_instance" "db_instance" {
  name             = "gcpdb"
  database_version = "MYSQL_8_0"
  settings {
    tier = "db-f1-micro"
  }
}

resource "google_sql_user" "db_user" {
  name     = var.db_username
  instance = google_sql_database_instance.db_instance.name
  password = var.db_password
}
```

variables.tf

```
variable "project" {}

variable "region" {}

variable "db_username" {}

variable "db_password" {}
```

outputs.tf

```
output "db_connection_name" {

  value =
google_sql_database_instance.db_instance.connection_name

}
```

terraform.tfvars

```
project      = "your-gcp-project-id"

region       = "us-central1"

db_username  = "admin"

db_password  = "securepassword"
```

}

Oracle Cloud: Database Setup Project

Directory Structure:

```
oracle-database-setup/
├── main.tf
├── variables.tf
├── outputs.tf
├── terraform.tfvars
```

Files:

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_database_autonomous_database" "db_instance" {

  compartment_id = var.compartment_id

  db_name        = "oracledb"

  admin_password = var.db_password

  cpu_core_count = 1

  data_storage_size_in_tbs = 1

}
```

variables.tf

```
variable "tenancy_ocid" {}

variable "user_ocid" {}

variable "fingerprint" {}

variable "private_key_path" {}

variable "region" {}

variable "compartment_id" {}

variable "db_password" {}
```

outputs.tf

```
output "db_id" {
  value = oci_database_autonomous_database.db_instance.id
}
```

terraform.tfvars

```
tenancy_ocid      = "your-tenancy-ocid"
user_ocid         = "your-user-ocid"
fingerprint       = "your-fingerprint"
private_key_path  = "~/.oci/private_key.pem"
region            = "us-ashburn-1"
compartment_id    = "your-compartment-id"
db_password       = "securepassword"
```

Usage Instructions

Navigate to the directory of the desired cloud project:

```
cd aws-database-setup  # Replace with azure-database-setup,
gcp-database-setup, or oracle-database-setup
```

Initialize Terraform:

```
terraform init
```

Validate the configuration:

```
terraform validate
```

Apply the configuration:

```
terraform apply
```

Clean up resources:

```
terraform destroy
```

Project 2. Database Sharding with Terraform

Set up a multi-region, multi-cloud, or sharded database infrastructure for horizontal scaling using Terraform.

This project demonstrates how to implement database sharding using Terraform across multiple clouds or regions to achieve horizontal scaling. Below is a detailed step-by-step guide on setting up database sharding in AWS, Azure, GCP, and Oracle using Terraform.

AWS Database Sharding with Terraform (Step-by-Step)

Step 1 Configure AWS Provider

Create a `main.tf` file and configure the AWS provider.

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

Step 2 Set Up Sharded Database (RDS) Instances

Define two database instances (shards) for the sharded setup in AWS RDS.

```
resource "aws_db_instance" "shard_1" {
  identifier         = "shard-1"
  engine             = "postgres"
  instance_class     = "db.t3.medium"
  allocated_storage  = 20
  storage_type       = "gp2"
  multi_az           = true
  db_subnet_group_name = aws_db_subnet_group.example.name
}
```

```
resource "aws_db_instance" "shard_2" {

  identifier         = "shard-2"

  engine             = "postgres"

  instance_class     = "db.t3.medium"

  allocated_storage  = 20

  storage_type       = "gp2"

  multi_az           = true

  db_subnet_group_name = aws_db_subnet_group.example.name

}


resource "aws_db_subnet_group" "example" {

  name       = "example"

  subnet_ids = [aws_subnet.subnet1.id,
aws_subnet.subnet2.id]

}


resource "aws_subnet" "subnet1" {

  vpc_id     = aws_vpc.example.id

  cidr_block = "10.0.1.0/24"

}


resource "aws_subnet" "subnet2" {
  vpc_id     = aws_vpc.example.id

  cidr_block = "10.0.2.0/24"

}


resource "aws_vpc" "example" {

  cidr_block = "10.0.0.0/16"

}
```

Step 3 Initialize and Apply Terraform Configuration

- Initialize and apply the configuration

```
terraform init

terraform apply
```

Azure Database Sharding with Terraform (Step-by-Step)

Step 1 Configure Azure Provider

Create a `main.tf` file and configure the Azure provider.

```
provider "azurerm" {

  features {}

}
```

Step 2 Set Up Sharded Database (Azure PostgreSQL) Instances

Define two PostgreSQL server instances in different regions.

```
resource "azurerm_postgresql_server" "shard_1" {

  name                = "shard-1-server"

  location            = "East US"

  resource_group_name = azurerm_resource_group.example.name

  administrator_login = "adminuser"

  administrator_login_password = "password123"

  version             = "11"

}


resource "azurerm_postgresql_server" "shard_2" {

  name                = "shard-2-server"

  location            = "West US"

  resource_group_name = azurerm_resource_group.example.name

  administrator_login = "adminuser"
  administrator_login_password = "password123"
  version                      = "11"
}


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

Step 3 Initialize and Apply Terraform Configuration

- Initialize and apply the configuration

```
terraform init
terraform apply
```

GCP Database Sharding with Terraform (Step-by-Step)

Step 1 Configure GCP Provider

Create a `main.tf` file and configure the GCP provider.

```
provider "google" {
  region = "us-central1"
}
```

Step 2 Set Up Sharded Database (GCP SQL PostgreSQL) Instances

Define two PostgreSQL database instances in different regions.

```
resource "google_sql_database_instance" "shard_1" {
  name             = "shard-1-db"
  database_version = "POSTGRES_13"
  region           = "us-central1"

  settings {
    tier = "db-f1-micro"
  }
}

resource "google_sql_database_instance" "shard_2" {
  name             = "shard-2-db"
  database_version = "POSTGRES_13"
  region           = "us-west1"


  settings {
    tier = "db-f1-micro"
  }
}
```

Step 3 Initialize and Apply Terraform Configuration

- Initialize and apply the configuration

```
terraform init
terraform apply
```

Oracle Database Sharding with Terraform (Step-by-Step)

Step 1 Configure Oracle Provider

Create a `main.tf` file and configure the Oracle provider.

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

Step 2 Set Up Sharded Database (OCI Autonomous Database) Instances

Define two Autonomous Database instances in Oracle Cloud.

```
resource "oci_db_autonomous_database" "shard_1" {

  db_name                = "shard1-db"

  compartment_id         =
oci_identity_compartment.example.id

  cpu_core_count         = 1

  db_workload            = "OLTP"

  admin_password         = "password123"

  db_unique_name         = "shard1-unique"

  subnet_id              = oci_core_subnet.example.id

}


resource "oci_db_autonomous_database" "shard_2" {

  db_name                = "shard2-db"

  compartment_id         =
oci_identity_compartment.example.id

  cpu_core_count         = 1

  db_workload            = "OLTP"
  admin_password       = "password123"

  db_unique_name       = "shard2-unique"

  subnet_id            = oci_core_subnet.example.id

}


resource "oci_core_subnet" "example" {

  compartment_id = oci_identity_compartment.example.id

  vcn_id         = oci_core_vcn.example.id

  cidr_block     = "10.0.0.0/24"

}


resource "oci_identity_compartment" "example" {

  name          = "example-compartment"

  compartment_id = "<root_compartment_id>"

}
```

Step 3 Initialize and Apply Terraform Configuration

- Initialize and apply the configuration

```
terraform init

terraform apply
```

Final Step Clean Up Resources

Once testing is done, you can clean up all the created resources using

```
terraform destroy
```

Project 3. Database Clustering with Terraform

Automate the setup of database clusters such as MySQL, PostgreSQL, or MongoDB with Terraform for high availability and load balancing.

Here's a step-by-step guide to setting up **Database Clustering with Terraform** for high availability and load balancing. We'll cover AWS, Azure, GCP, and Oracle separately, starting with prerequisites common to all.

Prerequisites

- **Terraform** installed on your local machine.
- Cloud provider accounts
  - AWS (with IAM user and access keys).
  - Azure (with a Service Principal).
  - GCP (with a service account and JSON key).
  - Oracle Cloud (with tenancy details).
- CLI tools for each cloud provider.
- Basic knowledge of Terraform syntax.

---

1. AWS Setting Up a MySQL Cluster with Terraform

AWS offers services like RDS and Aurora for clustering.

Steps

Initialize Terraform and AWS Provider
Configure Terraform to use AWS as the provider. Create a provider.tf

```
provider "aws" {

  region = "us-east-1"

}
```

Create VPC and Networking Define the VPC, subnets, and security groups

```
resource "aws_vpc" "main" {

  cidr_block = "10.0.0.0/16"

}


resource "aws_subnet" "subnet1" {

  vpc_id            = aws_vpc.main.id

  cidr_block        = "10.0.1.0/24"

  availability_zone = "us-east-1a"

}
```

Provision RDS Instances Create a primary and replica instance

```
resource "aws_db_instance" "primary" {

  allocated_storage    = 20
  engine              = "mysql"

  instance_class      = "db.t3.micro"

  name                = "primarydb"

  username            = "admin"

  password            = "password123"

  skip_final_snapshot  = true

}


resource "aws_db_instance" "replica" {

  replicate_source_db  = aws_db_instance.primary.id

  instance_class      = "db.t3.micro"

}
```

Apply Terraform Config

- Run terraform init.
- Run terraform apply.

---

2. Azure Setting Up a PostgreSQL Flexible Server

Azure supports clustering via Flexible Server for PostgreSQL.

Steps

Initialize Azure Provider
Create provider.tf with Azure details

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

Create Resource Group and Network

```
resource "azurerm_resource_group" "rg" {
  name     = "db-cluster-rg"
  location = "East US"
}
```

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

Create PostgreSQL Flexible Server Define primary and standby servers

```
resource "azurerm_postgresql_flexible_server" "primary" {
  name                = "postgres-primary"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location

  version             = "13"

  sku_name            = "Standard_B2s"

  storage_mb          = 32768

}
```

Add Load Balancer Configure Azure Load Balancer to route traffic.

Apply Terraform Config

- Run terraform init.
- Run terraform apply.

---

3. GCP Setting Up a MongoDB Cluster

GCP uses Compute Instances with Managed Instance Groups.

Steps

Initialize GCP Provider
Create provider.tf with GCP details

```
provider "google" {

  project = "your-project-id"

  region  = "us-central1"

}
```

Create VPC and Subnet

```
resource "google_compute_network" "vpc" {
  name = "db-cluster-network"
}

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

Provision Compute Instances for MongoDB Use an instance group for clustering

```
resource "google_compute_instance_template" "mongo_template" {
  name = "mongo-template"
  machine_type = "n1-standard-1"

  disk {
    boot = true
    auto_delete = true
    initialize_params {
      image = "ubuntu-os-cloud/ubuntu-2204-jammy-v20230913"
    }
  }
}

resource "google_compute_instance_group_manager" "mongo_group" {
  base_instance_name = "mongo"
  instance_template  = google_compute_instance_template.mongo_template.id
  target_size        = 3
}
```

Apply Terraform Config

- Run terraform init.
- Run terraform apply.

---

4. Oracle Cloud Setting Up MySQL Cluster

Oracle offers MySQL Database Service with Group Replication.

Steps

Initialize Oracle Provider
Create provider.tf

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

Create VCN and Subnets

```
resource "oci_core_vcn" "vcn" {
  cidr_block = "10.0.0.0/16"
  display_name = "db-vcn"
}

resource "oci_core_subnet" "subnet" {
  vcn_id = oci_core_vcn.vcn.id
  cidr_block = "10.0.1.0/24"
  display_name = "db-subnet"
}
```

Create MySQL Database Service Use Oracle MySQL Service

```
resource "oci_mysql_mysql_db_system" "mysql_cluster" {
  compartment_id = "your-compartment-id"
  availability_domain = "kNnvUS-ASHBURN-AD-1"
  shape_name = "VM.Standard.E3.1.Micro"
  admin_username = "admin"

  admin_password = "Password123"

}
```

Apply Terraform Config

- Run terraform init.
- Run terraform apply.

Conclusion

These steps help automate database clustering with high availability and load balancing.

You can further customize configurations like backup policies and monitoring.

Always validate Terraform plans before applying to ensure resource correctness.

Project 4. Managed Database with Terraform

Use Terraform to provision fully managed cloud databases, such as AWS RDS, Azure Database for MySQL/PostgreSQL, or GCP Cloud SQL, with automated backups, replication, and scaling.

AWS Provision AWS RDS with Terraform

Step 1 Initialize Terraform Project

- Create a directory for the project and navigate to it.

Initialize Terraform

```
terraform init
```

Step 2 Define the AWS Provider

Create a provider.tf file

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

Step 3 Configure an AWS RDS Instance

Create a main.tf file with RDS configuration

```
resource "aws_db_instance" "example" {
  allocated_storage    = 20
  engine               = "mysql"
  engine_version       = "8.0"
  instance_class       = "db.t2.micro"
  name                 = "mydatabase"
  username             = "admin"
  password             = "password123"
  backup_retention_period = 7
  multi_az             = true
  storage_encrypted    = true
  skip_final_snapshot  = true
}
```

Step 4 Apply Configuration

Run the commands

```
terraform plan

terraform apply
```

Azure Provision Azure Database for MySQL/PostgreSQL

Step 1 Initialize Terraform Project

Initialize Terraform

```
terraform init
```

Step 2 Define Azure Provider

Create a provider.tf file

```
provider "azurerm" {

  features {}

}
```

Step 3 Configure Azure Database

Create a main.tf file for the database

```
resource "azurerm_mysql_flexible_server" "example" {

  name                = "mydatabase"

  resource_group_name = azurerm_resource_group.example.name
  location            = azurerm_resource_group.example.location
  sku_name            = "Standard_B1ms"
  storage_mb          = 5120
  backup_retention_days = 7
  geo_redundant_backup_enabled = true

  administrator_login          = "admin"
  administrator_login_password = "password123"
}

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

Step 4 Apply Configuration

Run the commands

```
terraform plan
terraform apply
```

GCP Provision GCP Cloud SQL

Step 1 Initialize Terraform Project

Initialize Terraform

```
terraform init
```

Step 2 Define GCP Provider

Create a provider.tf file

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

Step 3 Configure Cloud SQL Instance

Create a main.tf file

```
resource "google_sql_database_instance" "example" {
  name             = "mydatabase"
  database_version = "MYSQL_8_0"
  region           = "us-central1"

  settings {
    tier         = "db-f1-micro"
    backup_configuration {
      enabled = true
    }
    availability_type = "REGIONAL"
  }
}

resource "google_sql_user" "example" {
  instance = google_sql_database_instance.example.name
  name     = "admin"
  password = "password123"
}
```

Step 4 Apply Configuration

Run the commands

```
terraform plan
terraform apply
```

Oracle Provision Oracle Autonomous Database

Step 1 Initialize Terraform Project

Initialize Terraform

```
terraform init
```

}

Step 2 Define Oracle Provider

Create a provider.tf file

```
provider "oci" {
  tenancy_ocid     = var.tenancy_ocid
  user_ocid        = var.user_ocid
  fingerprint      = var.fingerprint
  private_key_path = var.private_key_path
  region           = "us-ashburn-1"
}
```

Step 3 Configure Oracle Autonomous Database

Create a main.tf file

```
resource "oci_database_autonomous_database" "example" {
  compartment_id = var.compartment_ocid
  db_name        = "mydatabase"
  admin_password = "Password123!"

  cpu_core_count     = 1
  data_storage_size_in_tbs = 1
  db_workload        = "OLTP"
  is_auto_scaling_enabled = true
}
```

Step 4 Apply Configuration

Run the commands

```
terraform plan

terraform apply
```

Notes for All Platforms

- **Prerequisites**
  - Ensure proper Terraform CLI and provider setup.
  - Configure necessary environment variables or authentication files for each cloud provider.
- **Security** Avoid hardcoding sensitive information (e.g., passwords). Use Terraform variables or secret managers.
- **Backup/Restore** Verify automatic backup configurations for each provider.

Project 5. Terraform for Cloud-Native Storage Solutions

Automate the deployment of cloud-native storage solutions such as Amazon EFS, GCP Filestore, or Azure NetApp Files, integrating them with your infrastructure using Terraform.

Here’s a step-by-step guide to using Terraform to deploy cloud-native storage solutions on **AWS (EFS)**, **Azure (NetApp Files)**, **GCP (Filestore)**, and **Oracle Cloud (File Storage)**.

AWS Amazon Elastic File System (EFS)

Prerequisites

- AWS account
- Terraform installed locally
- AWS CLI installed and configured

Steps

1. **Create a Terraform Configuration File**

Create a file called main.tf and define the AWS provider

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

Define an EFS File System

```
resource "aws_efs_file_system" "efs" {
  lifecycle_policy {
    transition_to_ia = "AFTER_30_DAYS"
  }
  tags = {
    Name = "MyEFS"
  }
}
```

Create a Mount Target for EFS

```
resource "aws_efs_mount_target" "mount" {
  file_system_id = aws_efs_file_system.efs.id
  subnet_id      = "your-subnet-id"
  security_groups = ["your-security-group-id"]
}
```

Initialize Terraform

```
terraform init
```

Plan the Infrastructure

```
terraform plan
```

Apply the Configuration

```
terraform apply
```

Azure Azure NetApp Files

Prerequisites

- Azure account
- Terraform installed locally
- Azure CLI installed and configured

Steps

1. **Create a Terraform Configuration File**

Create a file called main.tf and define the Azure provider

```
provider "azurerm" {

  features {}
}
```

Create an Azure NetApp Account

```
resource "azurerm_netapp_account" "example" {
  name                = "example-anf-account"
  resource_group_name = "your-resource-group"
  location            = "East US"
}
```

Create a NetApp Pool

```
resource "azurerm_netapp_pool" "example" {
  name                = "example-pool"
  location            = azurerm_netapp_account.example.location
  resource_group_name = azurerm_netapp_account.example.resource_group_name
  service_level       = "Premium"
  size_in_tb          = 4
}
```

Create a Volume

```
resource "azurerm_netapp_volume" "example" {
  name                = "example-volume"
  resource_group_name = azurerm_netapp_account.example.resource_group_name
  location            = azurerm_netapp_account.example.location
  volume_path         = "/example-path"
  netapp_pool_name    = azurerm_netapp_pool.example.name
  service_level       = "Premium"
  usage_threshold     = 100
}
```

Initialize, Plan, and Apply Terraform

```
terraform init
terraform plan
terraform apply
```

GCP Filestore

Prerequisites

- GCP account
- Terraform installed locally
- gcloud CLI installed and configured

Steps

1. **Create a Terraform Configuration File**

Define the GCP provider

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

Create a Filestore Instance

```
resource "google_filestore_instance" "example" {
  name       = "example-instance"
  tier       = "STANDARD"
  region     = "us-central1"
  zone       = "us-central1-a"
  networks {
    network    = "default"
    reserved_ip_range = "10.0.0.0/29"
  }
  file_shares {
    name        = "example-share"
    capacity_gb = 1024
  }
}
```

Initialize, Plan, and Apply Terraform

```
terraform init

terraform plan

terraform apply
```

Oracle Cloud File Storage

Prerequisites

- Oracle Cloud account
- Terraform installed locally
- OCI CLI installed and configured

Steps

1. **Create a Terraform Configuration File**

Define the OCI provider

```
provider "oci" {

  tenancy_ocid     = "your-tenancy-ocid"

  user_ocid        = "your-user-ocid"

  fingerprint      = "your-fingerprint"

  private_key_path = "your-private-key-path"

  region           = "us-ashburn-1"

}
```

Create a File Storage File System

```
resource "oci_file_storage_file_system" "example" {

  compartment_id = "your-compartment-id"

  availability_domain = "your-availability-domain"

  display_name = "example-fs"

}
```

Create a Mount Target

```
resource "oci_file_storage_mount_target" "example" {

  compartment_id = "your-compartment-id"

  availability_domain = "your-availability-domain"

  subnet_id = "your-subnet-id"

  display_name = "example-mount"

}
```

Initialize, Plan, and Apply Terraform

```
terraform init

terraform plan

terraform apply
```

Notes

- Replace placeholders like your-subnet-id, your-resource-group, your-compartment-id, etc., with actual values from your cloud provider.

- Each cloud provider has unique configurations; review their Terraform documentation for advanced options.
- Use separate directories or workspaces for each provider to avoid configuration overlap.

This step-by-step approach ensures clarity and seamless integration of Terraform with various cloud-native storage solutions.

Project 6. Infrastructure for Data Warehouse

Set up and provision a cloud data warehouse solution using Terraform, such as Amazon Redshift or Google BigQuery, for data analysis and reporting purposes.

1. AWS Amazon Redshift

Step 1 Install Prerequisites

- Install Terraform.
- Install AWS CLI and configure with your credentials using aws configure.

Step 2 Initialize Terraform Directory

- Create a directory for your project.
- Inside the directory, create a main.tf file.

Step 3 Write Terraform Configuration

Provider Configuration

```
provider "aws" {

  region = "us-east-1" # Replace with your preferred region

}
```

VPC and Subnet

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

resource "aws_subnet" "redshift_subnet" {
  vpc_id            = aws_vpc.redshift_vpc.id
  cidr_block        = "10.0.1.0/24"
  availability_zone = "us-east-1a"
}
```

-

Security Group

```
resource "aws_security_group" "redshift_sg" {
  vpc_id = aws_vpc.redshift_vpc.id

  ingress {
    from_port   = 5439
    to_port     = 5439
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Redshift Cluster

```
resource "aws_redshift_cluster" "redshift_cluster" {
  cluster_identifier      = "my-redshift-cluster"
  node_type               = "dc2.large"
  master_username         = "adminuser"
  master_password         = "adminpassword"
  cluster_type            = "single-node"
  db_name                 = "mydatabase"
  vpc_security_group_ids   = [aws_security_group.redshift_sg.id]
  cluster_subnet_group_name = aws_subnet.redshift_subnet.id
}
```

Step 4 Initialize, Plan, and Apply

- Run terraform init.
- Run terraform plan to check your configuration.

- Run terraform apply and confirm with yes.

---

2. Azure Synapse Analytics

Step 1 Install Prerequisites

- Install Terraform.
- Install Azure CLI and authenticate with az login.

Step 2 Write Terraform Configuration

Provider Configuration

```
provider "azurerm" {

  features = {}

}
```

-

Resource Group

```
resource "azurerm_resource_group" "example" {

  name     = "example-resources"

  location = "East US"

}
```

Azure Synapse Workspace

```
resource "azurerm_synapse_workspace" "example" {

  name                              = "example-synapse"
  resource_group_name              = azurerm_resource_group.example.name
  location                         = azurerm_resource_group.example.location
  sql_administrator_login          = "adminuser"
  sql_administrator_login_password  = "adminpassword"
  storage_account_url              = azurerm_storage_account.example.primary_blob_endpoint
  storage_file_system_name         = "synapsestorage"
}
```

Storage Account (for Synapse Workspace)

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

Step 3 Initialize, Plan, and Apply

- Run terraform init.
- Run terraform plan.
- Run terraform apply.

---

3. GCP BigQuery

Step 1 Install Prerequisites

- Install Terraform.
- Install GCloud CLI and authenticate with gcloud auth application-default login.

Step 2 Write Terraform Configuration

Provider Configuration

```
provider "google" {

  project = "your-project-id"

  region  = "us-central1"

}
```

BigQuery Dataset

```
resource "google_bigquery_dataset" "example" {

  dataset_id = "example_dataset"

  location   = "US"

}
```

BigQuery Table

```
resource "google_bigquery_table" "example" {

  dataset_id = google_bigquery_dataset.example.dataset_id

  table_id   = "example_table"


  schema = <<EOT

  [
   {
     "name" "name",
     "type" "STRING",
     "mode" "REQUIRED"
   },
   {
     "name" "age",
     "type" "INTEGER",
     "mode" "NULLABLE"
   }
  ]
  EOT
}
```

Step 3 Initialize, Plan, and Apply

- Run terraform init.
- Run terraform plan.
- Run terraform apply.

---

4. Oracle Autonomous Data Warehouse

Step 1 Install Prerequisites

- Install Terraform.
- Install OCI CLI and configure credentials.

Step 2 Write Terraform Configuration

Provider Configuration

```
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-ashburn-1"
}
```

Autonomous Data Warehouse

```
resource "oci_database_autonomous_database" "example" {
  compartment_id = "your-compartment-id"
  db_name        = "exampleadb"
  admin_password = "Adminpassword123!"
  cpu_core_count = 1
  data_storage_size_in_tbs = 1
}
```

Step 3 Initialize, Plan, and Apply

- Run terraform init.

- Run terraform plan.
- Run terraform apply.

---

10. Specialized Infrastructure

Project 1. Multi-Tier Application Architecture

Overview

Design a multi-tier architecture on popular cloud platforms (AWS, Azure, GCP, and OCI) to isolate the front-end, back-end, and database layers. Automate the infrastructure setup using Terraform for efficient and reproducible deployments.

---

1. AWS Architecture

Architecture Overview

- **Front-End** EC2 instances behind an Application Load Balancer (ALB).
- **Back-End** Private EC2 instances.
- **Database Layer** Managed RDS instance in a private subnet.

Steps

Setup Terraform Provider

```
provider "aws" {

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

Create VPC and Subnets

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

resource "aws_subnet" "public" {
  vpc_id                  = aws_vpc.main.id
  cidr_block              = "10.0.1.0/24"
  map_public_ip_on_launch = true
}

resource "aws_subnet" "private" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.2.0/24"
}
```

Create Security Groups

```
resource "aws_security_group" "frontend" {
  vpc_id = aws_vpc.main.id
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Deploy EC2 Instances

```
resource "aws_instance" "frontend" {
  ami           = "ami-12345678"
  instance_type   = "t2.micro"
  subnet_id       = aws_subnet.public.id
  security_groups = [aws_security_group.frontend.name]
}
```

Setup Application Load Balancer

```
resource "aws_lb" "app_lb" {
  name              = "app-lb"
  internal          = false
  load_balancer_type = "application"
  security_groups    = [aws_security_group.frontend.id]
  subnets           = [aws_subnet.public.id]
}
```

Configure Backend and Database

```
resource "aws_instance" "backend" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.private.id
}


resource "aws_db_instance" "database" {
  engine            = "mysql"
  instance_class    = "db.t3.micro"
  allocated_storage  = 20
  username          = "admin"
  password          = "password"
  subnet_group      = aws_db_subnet_group.default.name
}
```

---

2. Azure Architecture

Architecture Overview

- **Front-End** Public App Service.
- **Back-End** Hosted in a Virtual Network (VNet).
- **Database Layer** Azure SQL Database.

Steps

Setup Terraform Provider

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

Create Resource Group and VNet

```
resource "azurerm_resource_group" "main" {
  name     = "multi-tier-app"
  location = "East US"
}

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

Deploy Front-End App Service

```
resource "azurerm_app_service_plan" "plan" {
  name                = "frontend-plan"
  resource_group_name = azurerm_resource_group.main.name
  location            = azurerm_resource_group.main.location
  sku {
    tier = "Free"
    size = "F1"
  }
}

resource "azurerm_app_service" "frontend" {
  name               = "frontend-app"
  resource_group_name = azurerm_resource_group.main.name
  location           = azurerm_resource_group.main.location
  app_service_plan_id = azurerm_app_service_plan.plan.id
}
```

Create Backend VMs

```
resource "azurerm_network_interface" "backend_nic" {
  name               = "backend-nic"
  resource_group_name = azurerm_resource_group.main.name
  location           = azurerm_resource_group.main.location
}

resource "azurerm_virtual_machine" "backend" {
  name                 = "backend-vm"
  resource_group_name   = azurerm_resource_group.main.name
  location             = azurerm_resource_group.main.location
  network_interface_ids = [azurerm_network_interface.backend_nic.id]
  vm_size              = "Standard_B1ls"
}
```

Deploy SQL Database

```
resource "azurerm_sql_server" "sql" {
  name                       = "sqlserver"
  resource_group_name          = azurerm_resource_group.main.name

  location                     = azurerm_resource_group.main.location

  administrator_login          = "sqladmin"

  administrator_login_password = "Password123!"

}

resource "azurerm_sql_database" "db" {

  name                = "appdb"

  resource_group_name = azurerm_resource_group.main.name

  server_name         = azurerm_sql_server.sql.name

}
```

---

3. GCP Architecture

Architecture Overview

- **Front-End** Managed Instance Group with a Load Balancer.
- **Back-End** Private Compute Engine instances.
- **Database Layer** Cloud SQL instance.

Steps

Setup Terraform Provider

```
provider "google" {

  project = "my-project-id"

  region  = "us-central1"

}
```

Create VPC and Subnets

```
resource "google_compute_network" "vpc" {
  name = "vpc-network"
}
resource "google_compute_subnetwork" "public" {
  name          = "public-subnet"
  network       = google_compute_network.vpc.id
  ip_cidr_range = "10.0.1.0/24"
}
```

Deploy Front-End

```
resource "google_compute_instance_group" "frontend" {
  name               = "frontend-group"
  base_instance_name = "frontend"
}
```

Configure Backend and Database

```
resource "google_compute_instance" "backend" {
  name         = "backend-instance"
  machine_type = "e2-micro"
  zone         = "us-central1-a"
}

resource "google_sql_database_instance" "db" {
  name = "database-instance"

}
```

---

4. OCI Architecture

Architecture Overview

- **Front-End** Load Balancer with Compute Instances.
- **Back-End** Private Compute Instances.
- **Database Layer** Autonomous Database.

Steps

Setup Terraform Provider

```
provider "oci" {

  region = "us-ashburn-1"

}
```

Create VCN and Subnets

```
resource "oci_core_vcn" "vcn" {

  cidr_block = "10.0.0.0/16"

}
```

Deploy Front-End Compute Instances

```
resource "oci_core_instance" "frontend" {

  display_name = "frontend-instance"

  shape        = "VM.Standard.E2.1.Micro"
}
```

Setup Backend and Autonomous Database

```
resource "oci_core_instance" "backend" {
  display_name = "backend-instance"
}


resource "oci_database_autonomous_database" "db" {
  db_name = "APPDB"
}
```

Project 2: Containerized Microservices Architecture

Introduction:

The **Containerized Microservices Architecture** project focuses on designing a scalable and resilient microservices environment using containerization and orchestration tools like Docker and Kubernetes. Each microservice is encapsulated within a Docker container and deployed to Kubernetes clusters, which can be provisioned on various cloud platforms such as AWS (EKS), Azure (AKS), Google Cloud (GKE), or Oracle Cloud (OKE). These microservices are designed to work independently but communicate with each other through REST APIs or messaging queues.

In this architecture, a cloud-managed PostgreSQL database serves as the central data storage, enabling consistent data access across different services. The system ensures high availability, load balancing, and fault tolerance using Kubernetes' built-in features, which include horizontal scaling, pod redundancy, and automatic failover.

The architecture is further optimized by incorporating **CI/CD pipelines**, which automate the building, testing, and deployment of the microservices. This ensures faster delivery of updates, with minimal downtime and human intervention. The infrastructure also integrates monitoring tools like Prometheus and Grafana for real-time performance tracking and logging, ensuring the health of the system is always visible and any potential issues are promptly addressed.

Terraform Configuration for Cloud Platforms:

The following Terraform configurations demonstrate how to set up this containerized architecture across multiple cloud platforms: **AWS, Azure, Google Cloud, and Oracle**.

AWS (Elastic Kubernetes Service - EKS + PostgreSQL)

Steps:

1. Set up an **EKS Cluster** for orchestration.
2. Create an **RDS PostgreSQL Database** for storage.
3. Deploy **Docker Containers** within Kubernetes pods.

# EKS Cluster

```
resource "aws_eks_cluster" "example" {
  name     = "example-eks-cluster"
  role_arn = aws_iam_role.eks_cluster_role.arn
  vpc_config {
    subnet_ids = aws_subnet.example.*.id
  }
}
```

# RDS PostgreSQL Database

```
resource "aws_db_instance" "example" {
  identifier = "example-postgres-db"
  engine     = "postgres"
  instance_class = "db.t3.micro"
  allocated_storage = 20
  db_name   = "exampledb"
  username  = "admin"

  password  = "password123"

}


# EKS Node Group

resource "aws_eks_node_group" "example" {

  cluster_name    = aws_eks_cluster.example.name

  node_role_arn   = aws_iam_role.eks_node_role.arn

  subnet_ids      = aws_subnet.example.*.id

  instance_types  = ["t3.micro"]

}
```

Azure (Azure Kubernetes Service - AKS + PostgreSQL)

Steps:

1. Set up an **AKS Cluster** for container orchestration.
2. Create an **Azure Database for PostgreSQL** for persistent data.
3. Deploy **Docker Containers** within AKS pods.

```
# AKS Cluster

resource "azurerm_kubernetes_cluster" "example" {

  name                = "example-aks-cluster"

  location            = "East US"

  resource_group_name = azurerm_resource_group.example.name

  default_node_pool {
    name       = "default"
    node_count = 1
    vm_size    = "Standard_B1s"
  }
}

# Azure Database for PostgreSQL
resource "azurerm_postgresql_server" "example" {
  name                         = "examplepgserver"
  location                     = "East US"
  resource_group_name          = azurerm_resource_group.example.name
  administrator_login          = "adminuser"
  administrator_login_password = "password123"
  version                      = "11"
  sku_name                     = "B_Gen5_1"
}

# AKS Node Pool
resource "azurerm_kubernetes_cluster_node_pool" "example" {
  name                  = "default"
  kubernetes_cluster_id = azurerm_kubernetes_cluster.example.id
  node_count            = 1
  vm_size               = "Standard_B1s"
}
```

Google Cloud (Google Kubernetes Engine - GKE + PostgreSQL)

Steps:

1. Set up a **GKE Cluster** for orchestration.
2. Create a **Cloud SQL PostgreSQL** instance for data storage.
3. Deploy **Docker Containers** within GKE pods.

```
# GKE Cluster

resource "google_container_cluster" "example" {

  name     = "example-gke-cluster"

  location = "us-central1"

  initial_node_count = 1

  node_config {

    machine_type = "e2-medium"

  }

}


# Cloud SQL PostgreSQL

resource "google_sql_database_instance" "example" {

  name             = "example-sql-instance"

  region           = "us-central1"

  database_version = "POSTGRES_13"

  tier             = "db-f1-micro"
}


# GKE Node Pool
resource "google_container_node_pool" "example" {
  name               = "default-node-pool"
  cluster            = google_container_cluster.example.name
  initial_node_count = 1
  node_config {
    machine_type = "e2-medium"
  }
}
```

Oracle Cloud (Oracle Kubernetes Engine - OKE + PostgreSQL)

Steps:

1. Set up an **OKE Cluster** for container orchestration.
2. Create an **Autonomous Database** for PostgreSQL data storage.
3. Deploy **Docker Containers** in Oracle Kubernetes Engine.

```
# OKE Cluster
resource "oci_containerengine_cluster" "example" {
  name = "example-oke-cluster"
  compartment_id = var.compartment_id
  vcn_id = oci_core_virtual_network.example.id
}


# Autonomous Database for PostgreSQL

resource "oci_database_autonomous_database" "example" {

  db_name = "exampledb"

  compartment_id = var.compartment_id

  cpu_core_count = 1

  db_workload = "OLTP"

  admin_password = "password123"

}


# OKE Node Pool

resource "oci_containerengine_node_pool" "example" {

  cluster_id = oci_containerengine_cluster.example.id

  name       = "example-node-pool"

  node_shape = "VM.Standard2.1"

  quantity   = 1

}
```

Conclusion:

This **Containerized Microservices Architecture** provides a robust foundation for deploying microservices at scale in any of the leading cloud platforms (AWS, Azure, GCP, Oracle). Each cloud platform utilizes Kubernetes for orchestration, cloud-managed PostgreSQL for data persistence, and Terraform for automated provisioning of infrastructure. This project highlights the power of containerization and cloud-native services in building scalable, resilient, and easily manageable microservices-based applications.

Project 3. Infrastructure for Edge Computing

Edge computing minimizes latency and reduces bandwidth use by processing data closer to IoT devices. This project demonstrates how to automate the creation of infrastructure for edge computing applications using **Terraform** across various cloud platforms. The setup integrates **IoT devices**, **edge servers**, and **cloud services** to ensure scalability, efficiency, and seamless communication.

Steps Overview

- **AWS** Set up S3 for IoT data storage, EC2 instances for edge servers, and IoT Core for device communication.
- **Azure** Configure IoT Hub, Virtual Machines, and networking resources.
- **Google Cloud Platform (GCP)** Use Cloud Storage, Compute Engine, and IoT Core Registry for edge processing.
- **Oracle Cloud Infrastructure (OCI)** Provision Object Storage, Compute Instances, and networking for edge computing.

---

1. AWS Infrastructure Setup

Step 1 Prerequisites

- Install **Terraform**.
- Configure the **AWS CLI** with access keys.
- Enable necessary services (e.g., IoT Core, EC2, S3).

Step 2 Define the Project Directory

- Create a directory for Terraform configuration files.
- Add a `main.tf` file for resource definitions.

Step 3 Write Terraform Configuration

```
provider "aws" {

  region = "us-east-1"

}



resource "aws_s3_bucket" "iot_data" {

  bucket = "iot-edge-data-storage"

}



resource "aws_instance" "edge_server" {

  ami           = "ami-0c2b8ca1dad447f8a"

  instance_type = "t2.micro"

  tags = {

    Name = "EdgeServer"

  }

}
```

```
resource "aws_iot_topic_rule" "iot_rule" {

  name        = "EdgeRule"

  description = "Rule for Edge IoT integration"

  sql         = "SELECT * FROM 'iot/edge'"

  sql_version = "2016-03-23"

}
```

Step 4 Initialize and Apply

- Run `terraform init` to initialize Terraform.
- Execute `terraform apply` to create the resources.

---

2. Azure Infrastructure Setup

Step 1 Prerequisites

- Install **Terraform**.
- Configure the **Azure CLI** and log in.
- Enable necessary services like IoT Hub and Virtual Machines.

Step 2 Write Terraform Configuration

```
provider "azurerm" {

  features {}

}
```

```
resource "azurerm_iothub" "iot_hub" {
  name                = "edgeIotHub"
  resource_group_name = "EdgeResourceGroup"
  location            = "East US"
  sku {
    name     = "S1"
    capacity = 1
  }
}

resource "azurerm_virtual_machine" "edge_vm" {
  name                = "EdgeVM"
  location            = "East US"
  resource_group_name = "EdgeResourceGroup"
  network_interface_ids =
[azurerm_network_interface.edge_nic.id]
  vm_size             = "Standard_B1s"

  os_profile {
    computer_name  = "EdgeVM"
    admin_username = "adminuser"
    admin_password = "SecurePassword123"
  }

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

  storage_os_disk {
    name              = "osdisk"
    caching           = "ReadWrite"
    create_option     = "FromImage"
    managed_disk_type = "Standard_LRS"
  }
}
```

Step 3 Initialize and Apply

- Run `terraform init` and `terraform apply`.

---

3. Google Cloud Platform (GCP) Setup

Step 1 Prerequisites

- Install **Terraform**.
- Configure the **gcloud CLI** with a service account key.
- Enable APIs for IoT Core, Compute Engine, and Cloud Storage.

Step 2 Write Terraform Configuration

```
provider "google" {

  project = "your-gcp-project-id"

  region  = "us-central1"

}


resource "google_storage_bucket" "iot_bucket" {

  name     = "iot-edge-storage"

  location = "US"

}


resource "google_compute_instance" "edge_instance" {

  name         = "edge-instance"

  machine_type = "e2-micro"

  zone         = "us-central1-a"


  boot_disk {

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

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

resource "google_cloudiot_registry" "iot_registry" {
  name   = "edge-registry"
  region = "us-central1"

  mqtt_config {
    mqtt_enabled_state = "MQTT_ENABLED"
  }
}
```

Step 3 Initialize and Apply

- Run `terraform init` and `terraform apply`.

---

4. Oracle Cloud Infrastructure (OCI) Setup

Step 1 Prerequisites

- Install **Terraform**.
- Configure the **OCI CLI** with API keys.
- Enable services like Object Storage and Compute.

Step 2 Write Terraform Configuration

```
provider "oci" {

  tenancy_ocid     = "your-tenancy-ocid"

  user_ocid        = "your-user-ocid"

  fingerprint      = "your-key-fingerprint"

  private_key_path = "~/.oci/oci_api_key.pem"

  region           = "us-ashburn-1"

}


resource "oci_objectstorage_bucket" "iot_bucket" {

  namespace       = "your-namespace"

  name            = "iot-edge-storage"

  compartment_id  = "your-compartment-ocid"

}


resource "oci_core_instance" "edge_instance" {
  availability_domain = "your-availability-domain"
  compartment_id      = "your-compartment-ocid"
  shape               = "VM.Standard2.1"

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

  source_details {
    source_type = "image"
    source_id   = "your-image-ocid"
  }

  metadata = {
    ssh_authorized_keys = "your-ssh-public-key"
  }
}
```

Step 3 Initialize and Apply

- Run `terraform init` and `terraform apply`.

Project 4 Blockchain Infrastructure Setup

Objective Use Terraform to set up infrastructure for blockchain networks such as Ethereum nodes, Hyperledger, or other blockchain technologies.

---

1. AWS Blockchain Infrastructure Setup with Terraform

Steps

Install Prerequisites

- Install **Terraform** from the official site.
- Install and configure **AWS CLI** using `aws configure`.

Create IAM Role

- Ensure the IAM role has EC2 and S3 full access permissions.

Terraform Configuration

Provider

```
provider "aws" {

  region = "us-east-1"

}
```

Key Pair for EC2

```
resource "aws_key_pair" "blockchain_key" {

  key_name   = "blockchain-key"
  public_key = file("~/.ssh/id_rsa.pub")
}
```

Security Group

```
resource "aws_security_group" "blockchain_sg" {
  name        = "blockchain-sg"
  description = "Allow blockchain traffic"

  ingress {
    from_port   = 30303
    to_port     = 30303
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

EC2 Instance

```
resource "aws_instance" "blockchain_node" {
  ami             = "ami-0c02fb55956c7d316" # Amazon Linux 2 AMI
  instance_type   = "t2.medium"
  key_name        = aws_key_pair.blockchain_key.key_name
  security_groups = [aws_security_group.blockchain_sg.name]

  user_data = <<-EOF
    #!/bin/
    sudo yum update -y
    curl -fsSL https//get.docker.com -o get-docker.sh
    sh get-docker.sh
    sudo systemctl start docker

    sudo docker pull ethereum/client-go

    sudo docker run -d --name ethereum-node -p 3030330303
ethereum/client-go

  EOF

}
```

Deployment

```
terraform init

terraform apply
```

---

2. Azure Blockchain Infrastructure Setup with Terraform

Steps

Install Prerequisites

- Install **Azure CLI** and authenticate using `az login`.
- Install Terraform.

Terraform Configuration

Provider

```
provider "azurerm" {

  features {}

}
```

Resource Group

```
resource "azurerm_resource_group" "blockchain_rg" {

  name     = "blockchain-network"

  location = "East US"

}
```

Virtual Machine

```
resource "azurerm_linux_virtual_machine" "blockchain_vm" {

  name                 = "blockchain-node"

  resource_group_name  = 
azurerm_resource_group.blockchain_rg.name

  location             = 
azurerm_resource_group.blockchain_rg.location

  size                 = "Standard_DS2_v2"

  admin_username       = "azureuser"
  admin_ssh_key {

    username   = "azureuser"

    public_key = file("~/.ssh/id_rsa.pub")

  }

  os_disk {

    caching              = "ReadWrite"

    storage_account_type = "Standard_LRS"

  }

  network_interface_ids =
[azurerm_network_interface.blockchain_nic.id]

  source_image_reference {

    publisher = "Canonical"

    offer     = "UbuntuServer"

    sku       = "18.04-LTS"

    version   = "latest"

  }

  custom_data = <<-EOT

    #!/bin/

    sudo apt update -y

    sudo apt install docker.io -y

    sudo docker pull ethereum/client-go
    sudo docker run -d --name ethereum-node -p 3030330303
ethereum/client-go

  EOT

}
```

Deployment

```
terraform init

terraform apply
```

---

3. GCP Blockchain Infrastructure Setup with Terraform

Steps

Set Up the Environment

- Authenticate with Google Cloud using `gcloud auth application-default login`.
- Install Terraform.

Terraform Configuration

Provider

```
provider "google" {

  project = "<YOUR_PROJECT_ID>"
  region  = "us-central1"
}
```

VM Instance

```
resource "google_compute_instance" "blockchain_node" {
  name         = "blockchain-node"
  machine_type = "e2-medium"
  zone         = "us-central1-a"
  boot_disk {
    initialize_params {
      image = "ubuntu-2004-focal-v20230606"
    }
  }
  network_interface {
    network       = "default"
    access_config {}
  }
  metadata_startup_script = <<-EOT
    #!/bin/
    sudo apt update -y
    sudo apt install docker.io -y

    sudo docker pull hyperledger/fabric-peer

    sudo docker run -d --name hyperledger-peer -p 70517051
hyperledger/fabric-peer

  EOT

}
```

Deployment

```
terraform init

terraform apply
```

---

4. Oracle Cloud Blockchain Infrastructure Setup with Terraform

Steps

Set Up Oracle CLI

- Authenticate using `oci setup config`.

Terraform Configuration

Provider

```
provider "oci" {
  tenancy_ocid      = "<TENANCY_OCID>"

  user_ocid         = "<USER_OCID>"

  fingerprint       = "<FINGERPRINT>"

  private_key_path  = "~/.oci/oci_api_key.pem"

  region            = "us-ashburn-1"

}
```

Compute Instance

```
resource "oci_core_instance" "blockchain_node" {

  availability_domain = "<AD>"

  compartment_id      = "<COMPARTMENT_ID>"

  display_name        = "blockchain-node"

  shape               = "VM.Standard.E2.1.Micro"


  create_vnic_details {

    subnet_id        = "<SUBNET_ID>"

    assign_public_ip = true

  }

  metadata = {

    ssh_authorized_keys = file("~/.ssh/id_rsa.pub")
    user_data = <<-EOT
      #!/bin/
      sudo yum update -y
      sudo yum install docker -y
      sudo systemctl start docker
      sudo docker pull hyperledger/fabric-ca
      sudo docker run -d --name fabric-ca -p 70547054
hyperledger/fabric-ca
    EOT
  }
  source_details {
    source_type = "image"
    source_id   = "<IMAGE_ID>"
  }
}
```

Deployment

```
terraform init
terraform apply
```

Common Tips

- Customize regions, instance sizes, and configurations as needed.
- Use `terraform plan` to preview changes before applying.
- Secure sensitive credentials using environment variables or a secret manager.

Project 5. AI and ML Model Deployment Infrastructure

Objective
Set up infrastructure using Terraform to host and deploy AI/ML models on platforms such as AWS SageMaker, GCP AI Platform, or Azure ML Studio.

Blockchain Infrastructure Setup Using Terraform

1. AWS Blockchain Infrastructure Setup

Objective Deploy an Ethereum node or Hyperledger network on AWS.

Steps

Install Prerequisites

- Install **Terraform** from the official site.
- Configure **AWS CLI** with `aws configure`.

Create IAM Role

- Use an IAM role with full access to EC2 and S3.

Define Terraform Configuration

Provider Configuration

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

Key Pair for EC2

```
resource "aws_key_pair" "blockchain_key" {
  key_name   = "blockchain-key"
  public_key = file("~/.ssh/id_rsa.pub")
}
```

Security Group

```
resource "aws_security_group" "blockchain_sg" {
  name        = "blockchain-sg"
  description = "Allow blockchain traffic"

  ingress {
    from_port   = 30303
    to_port     = 30303
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]

  }


  ingress {

    from_port   = 22

    to_port     = 22

    protocol    = "tcp"

    cidr_blocks = ["0.0.0.0/0"]

  }


  egress {

    from_port   = 0

    to_port     = 0

    protocol    = "-1"

    cidr_blocks = ["0.0.0.0/0"]

  }

}
```

EC2 Instance

```
resource "aws_instance" "blockchain_node" {
  ami             = "ami-0c02fb55956c7d316" # Amazon Linux 2 AMI
  instance_type   = "t2.medium"
  key_name        = aws_key_pair.blockchain_key.key_name
  security_groups = [aws_security_group.blockchain_sg.name]

  user_data = <<-EOF
    #!/bin/
    sudo yum update -y
    curl -fsSL https//get.docker.com -o get-docker.sh
    sh get-docker.sh
    sudo systemctl start docker
    sudo docker pull ethereum/client-go
    sudo docker run -d --name ethereum-node -p 3030330303
ethereum/client-go
  EOF
}
```

Initialize and Deploy

```
terraform init
terraform apply
```

---

2. Azure Blockchain Infrastructure Setup

Steps

Install Prerequisites

- Install **Azure CLI** and authenticate with `az login`.
- Install **Terraform**.

Define Terraform Configuration

Provider Configuration

```
provider "azurerm" {

  features {}

}
```

Resource Group

```
resource "azurerm_resource_group" "blockchain_rg" {

  name     = "blockchain-network"

  location = "East US"

}
```

Virtual Machine

```
resource "azurerm_linux_virtual_machine" "blockchain_vm" {

  name                = "blockchain-node"

  resource_group_name =
azurerm_resource_group.blockchain_rg.name

  location            =
azurerm_resource_group.blockchain_rg.location

  size                = "Standard_DS2_v2"

  admin_username      = "azureuser"


  admin_ssh_key {

    username   = "azureuser"

    public_key = file("~/.ssh/id_rsa.pub")

  }


  os_disk {

    caching              = "ReadWrite"

    storage_account_type = "Standard_LRS"

  }
  network_interface_ids =
[azurerm_network_interface.blockchain_nic.id]


  source_image_reference {

    publisher = "Canonical"

    offer     = "UbuntuServer"

    sku       = "18.04-LTS"

    version   = "latest"

  }


  custom_data = <<-EOF

    #!/bin/

    sudo apt update -y

    sudo apt install docker.io -y

    sudo docker pull ethereum/client-go

    sudo docker run -d --name ethereum-node -p 3030330303
ethereum/client-go

  EOF

}
```

Deploy the Configuration

```
terraform init

terraform apply
```

---

3. GCP Blockchain Infrastructure Setup

Steps

Set up Environment

- Authenticate using `gcloud auth application-default login`.
- Install **Terraform**.

Define Terraform Configuration

Provider Configuration

```
provider "google" {

  project = "<YOUR_PROJECT_ID>"

  region  = "us-central1"

}
```

VM Instance

```
resource "google_compute_instance" "blockchain_node" {

  name         = "blockchain-node"
  machine_type = "e2-medium"
  zone         = "us-central1-a"

  boot_disk {
    initialize_params {
      image = "ubuntu-2004-focal-v20230606"
    }
  }

  network_interface {
    network       = "default"
    access_config {}
  }

  metadata_startup_script = <<-EOT
    #!/bin/
    sudo apt update -y
    sudo apt install docker.io -y
    sudo docker pull hyperledger/fabric-peer
    sudo docker run -d --name hyperledger-peer -p 70517051
hyperledger/fabric-peer
  EOT
```

```
}
```

Initialize and Deploy

```
terraform init

terraform apply
```

---

4. Oracle Cloud Blockchain Infrastructure Setup

Steps

Set Up Environment

- Authenticate using `oci setup config`.

Define Terraform Configuration

Provider Configuration

```
provider "oci" {

  tenancy_ocid     = "<TENANCY_OCID>"

  user_ocid        = "<USER_OCID>"

  fingerprint      = "<FINGERPRINT>"

  private_key_path = "~/.oci/oci_api_key.pem"

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

Compute Instance

```
resource "oci_core_instance" "blockchain_node" {
  availability_domain = "<AD>"
  compartment_id      = "<COMPARTMENT_ID>"
  display_name        = "blockchain-node"
  shape               = "VM.Standard.E2.1.Micro"

  create_vnic_details {
    subnet_id        = "<SUBNET_ID>"
    assign_public_ip = true
  }

  metadata = {
    ssh_authorized_keys = file("~/.ssh/id_rsa.pub")
    user_data = <<-EOT
      #!/bin/
      sudo yum update -y
      sudo yum install docker -y
      sudo systemctl start docker

      sudo docker pull hyperledger/fabric-ca

      sudo docker run -d --name fabric-ca -p 70547054
hyperledger/fabric-ca

    EOT

  }


  source_details {

    source_type = "image"

    source_id   = "<IMAGE_ID>"

  }

}
```

Deploy the Configuration

```
terraform init

terraform apply
```

Common Tips

- Adjust configurations (e.g., region, instance size) based on requirements.
- Use `terraform plan` to preview changes.

- Secure sensitive data using environment variables or secret managers.

Project 6. Machine Learning Infrastructure

Create an infrastructure for machine learning projects on the cloud, including provisioning resources for training models, managing data storage, and deploying models for inference. Creating machine learning infrastructure on different cloud platforms using Terraform involves provisioning resources for training, managing data storage, and deploying models for inference. Below are step-by-step instructions tailored for AWS, Azure, GCP, and Oracle Cloud. Each section includes a separate Terraform implementation guide.

AWS Machine Learning Infrastructure

1. Install Terraform

Ensure Terraform is installed and configured. Set up AWS CLI with IAM credentials.

Define Provider

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

Create S3 Bucket for Data Storage

```
resource "aws_s3_bucket" "ml_bucket" {
bucket = "ml-data-storage-bucket"
acl = "private"
}
```

Provision EC2 Instance for Training

```
resource "aws_instance" "ml_training_instance" {
ami = "ami-0c02fb55956c7d316" # Example AMI (Ubuntu)
instance_type = "ml.m5.large"
tags = {
Name = "ML-Training-Instance"
}
}
```

Deploy Model with Lambda

```
resource "aws_lambda_function" "ml_inference_function" {
filename = "inference.zip"
function_name = "ml-inference"
role = aws_iam_role.lambda_exec.arn
handler = "app.lambda_handler"
runtime = "python3.9"
}
```

Output Resources

```
output "s3_bucket_name" {
value = aws_s3_bucket.ml_bucket.bucket
}
```

Azure Machine Learning Infrastructure

1. Install Terraform

Ensure Terraform is installed and configured with Azure CLI.

Define Provider

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

Create Azure Storage Account for Data

```
resource "azurerm_storage_account" "ml_storage" {
name = "mlstorageaccount"
resource_group_name = "ml-resources"
location = "eastus"
account_tier = "Standard"
account_replication_type = "LRS"
}
```

Provision VM for Training

```
resource "azurerm_virtual_machine" "ml_training_vm" {
name = "ml-training-vm"
resource_group_name = "ml-resources"
location = "eastus"
network_interface_ids = [azurerm_network_interface.ml_nic.id]
vm_size = "Standard_D2_v2"
storage_os_disk {
name = "osdisk"
caching = "ReadWrite"
create_option = "FromImage"

managed_disk_type = "Standard_LRS"

}

os_profile {

computer_name = "mltraining"

admin_username = "azureuser"

}

os_profile_linux_config {

disable_password_authentication = false

}

}
```

Deploy Model with Azure Functions

```
resource "azurerm_function_app" "ml_function_app" {

name = "ml-inference-function"

location = "eastus"

resource_group_name = "ml-resources"

storage_account_name = azurerm_storage_account.ml_storage.name

storage_account_access_key = azurerm_storage_account.ml_storage.primary_access_key

app_service_plan_id = azurerm_app_service_plan.ml_plan.id

}
```

GCP Machine Learning Infrastructure

1. Install Terraform

Ensure Terraform is installed and configured with gcloud credentials.

Define Provider

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

Create Cloud Storage Bucket

```
resource "google_storage_bucket" "ml_bucket" {
name = "ml-data-storage"
location = "US"
}
```

Provision Compute Engine for Training

```
resource "google_compute_instance" "ml_training_instance" {
name = "ml-training-instance"
machine_type = "n1-standard-4"
zone = "us-central1-a"
boot_disk {
initialize_params {
image = "debian-cloud/debian-10"
}
}
network_interface {
network = "default"
}
}
```

Deploy Model with Cloud Functions

```
resource "google_cloudfunctions_function" "ml_inference_function" {
name = "ml-inference-function"
runtime = "python39"
entry_point = "main"
source_archive_bucket = google_storage_bucket.ml_bucket.name
source_archive_object = "function_source.zip"
}
```

Oracle Cloud Machine Learning Infrastructure

1. Install Terraform

Ensure Terraform is installed and configured with OCI CLI.

Define Provider

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

Create Object Storage Bucket

```
resource "oci_objectstorage_bucket" "ml_bucket" {
name = "ml-data-storage"
namespace = "your_namespace"
compartment_id = "your_compartment_id"
}
```

Provision Compute Instance for Training

```
resource "oci_core_instance" "ml_training_instance" {
availability_domain = "your_ad"
compartment_id = "your_compartment_id"
shape = "VM.Standard2.1"
create_vnic_details {
subnet_id = "your_subnet_id"
}
metadata = {
ssh_authorized_keys = "your_ssh_key"
}
}
```

Deploy Model with Functions

```
resource "oci_functions_function" "ml_inference_function" {
display_name = "ml-inference"
application_id = "your_application_id"
memory_in_mbs = 128
image = "your_image"
}
```

Each step in the above guides can be expanded further to suit specific project requirements. Once complete, run the following Terraform commands for each platform

- terraform init
- terraform plan
- terraform apply

Project 7. AI-Powered Chatbot Infrastructure

Deploy an AI-powered chatbot infrastructure using **Terraform** across major cloud platforms AWS, Azure, GCP, and Oracle Cloud. Each implementation includes setting up the required resources for hosting, storing, and managing chatbot data.

AWS AI-Powered Chatbot Infrastructure

Step 1 Prerequisites

- Install **Terraform** and **AWS CLI**.

Configure AWS credentials

```
aws configure
```

-

Step 2 Create S3 Bucket for Terraform State

```
provider "aws" {

  region = "us-west-2"

}


resource "aws_s3_bucket" "terraform_state" {

  bucket = "chatbot-infra-terraform-state"

}
```

Step 3 Define Variables

Create `variables.tf`

```
variable "region" {

  default = "us-west-2"

}


variable "instance_type" {

  default = "t2.micro"

}
```

Step 4 Provision Resources

Create `main.tf`

```
resource "aws_ec2_instance" "chatbot_server" {

  ami           = "ami-0abcdef1234567890" # Replace with a valid
AMI ID

  instance_type = var.instance_type

  tags = {

    Name = "ChatbotServer"
  }
}

resource "aws_s3_bucket" "chatbot_data" {
  bucket = "chatbot-data-storage"
}
```

Step 5 Apply Configuration

```
terraform init
terraform plan
terraform apply
```

Azure AI-Powered Chatbot Infrastructure

Step 1 Prerequisites

- Install **Terraform** and **Azure CLI**.

Log in to Azure

```
az login
```

-

Step 2 Define Variables

Create `variables.tf`

```
variable "location" {
  default = "eastus"
}

variable "vm_size" {
  default = "Standard_B1s"
}
```

Step 3 Provision Resources

Create `main.tf`

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

resource "azurerm_resource_group" "chatbot_rg" {
  name     = "chatbot-resource-group"
  location = var.location
```

```
}


resource "azurerm_virtual_machine" "chatbot_vm" {

  name                  = "chatbot-vm"

  location              =
azurerm_resource_group.chatbot_rg.location

  resource_group_name   = azurerm_resource_group.chatbot_rg.name

  vm_size               = var.vm_size


  os_profile {

    computer_name  = "chatbotvm"

    admin_username = "adminuser"

    admin_password = "P@ssw0rd1234!"

  }


  os_profile_linux_config {}


  storage_image_reference {

    publisher = "Canonical"

    offer     = "UbuntuServer"

    sku       = "18.04-LTS"

    version   = "latest"
  }


  storage_os_disk {

    name              = "chatbot-os-disk"

    caching           = "ReadWrite"

    create_option     = "FromImage"

    managed_disk_type = "Standard_LRS"

  }


  network_interface_ids = [

    azurerm_network_interface.chatbot_nic.id,

  ]

}


resource "azurerm_network_interface" "chatbot_nic" {

  name                = "chatbot-nic"

  location            =
azurerm_resource_group.chatbot_rg.location

  resource_group_name = azurerm_resource_group.chatbot_rg.name

}
```

Step 4 Apply Configuration

```
terraform init

terraform plan

terraform apply
```

GCP AI-Powered Chatbot Infrastructure

Step 1 Prerequisites

- Install **Terraform** and **Google Cloud CLI**.

Authenticate

```
gcloud auth application-default login
```

-

Step 2 Create Variables

Create `variables.tf`

```
variable "project_id" {}

variable "region" {

  default = "us-central1"

}
```

Step 3 Provision Resources

Create `main.tf`

```
provider "google" {

  project = var.project_id

  region  = var.region

}


resource "google_compute_instance" "chatbot_vm" {

  name         = "chatbot-vm"

  machine_type = "n1-standard-1"

  zone         = "us-central1-a"


  boot_disk {

    initialize_params {

      image = "debian-cloud/debian-11"

    }

  }


  network_interface {

    network       = "default"
    access_config {}
  }
}

resource "google_storage_bucket" "chatbot_data" {
  name     = "chatbot-data-storage"
  location = var.region
}
```

Step 4 Apply Configuration

```
terraform init
terraform plan
terraform apply
```

Oracle Cloud AI-Powered Chatbot Infrastructure

Step 1 Prerequisites

- Install **Terraform** and **OCI CLI**.
- Generate an API key for Terraform.

Step 2 Define Variables

Create `variables.tf`

```
variable "tenancy_ocid" {}

variable "compartment_ocid" {}

variable "region" {}
```

Step 3 Provision Resources

Create `main.tf`

```
provider "oci" {

  tenancy_ocid     = var.tenancy_ocid

  user_ocid        = "<USER_OCID>"

  private_key_path = "<PRIVATE_KEY_PATH>"

  region           = var.region

}


resource "oci_core_instance" "chatbot_vm" {

  compartment_id = var.compartment_ocid

  availability_domain = "UocmPHX-AD-1"

  shape = "VM.Standard2.1"
  create_vnic_details {

    subnet_id = "<SUBNET_OCID>"

  }


  metadata = {

    ssh_authorized_keys = "<YOUR_SSH_PUBLIC_KEY>"

  }

}


resource "oci_objectstorage_bucket" "chatbot_data" {

  name           = "chatbot-data-storage"

  compartment_id = var.compartment_ocid

  storage_tier   = "Standard"

}
```

Step 4 Apply Configuration

```
terraform init

terraform plan

terraform apply
```

Common Considerations

- Use Terraform **modules** for reusable configurations.
- Enable **monitoring** and **logging** for insights.
- Configure **security groups** or firewall rules to restrict access.

10. Backup, Recovery & Automation

Project 1. Cloud Backup and Recovery

Creating a **Cloud Backup and Recovery** project using **Terraform** for each major cloud provider (AWS, Azure, GCP, and Oracle) involves different steps due to the unique resources and services offered by each provider. Below is a step-by-step guide for each.

General Steps

1. Define project objectives Automate cloud backup and recovery with Terraform.
2. Choose the cloud provider AWS, Azure, GCP, or Oracle.
3. Install prerequisites Terraform CLI, provider-specific CLI tools (e.g., AWS CLI).
4. Implement Terraform code specific to each provider.

AWS Backup and Recovery

1. **Set up environment**
    - Install AWS CLI and Terraform.

Configure AWS credentials

```
aws configure
```

Create S3 bucket for backup storage
Define the bucket in a main.tf file

```
resource "aws_s3_bucket" "backup_bucket" {

  bucket = "my-backup-bucket"

  versioning {

    enabled = true

  }

  lifecycle {

    prevent_destroy = true

  }

}
```

Define EC2 backup plan using AWS Backup
Add backup vault and plans

```
resource "aws_backup_vault" "backup_vault" {

  name        = "my_backup_vault"

  kms_key_arn = aws_kms_key.my_key.arn

}
```

```
resource "aws_backup_plan" "backup_plan" {

  name = "my_backup_plan"

  rule {
    rule_name         = "daily_backup"

    target_vault_name = aws_backup_vault.backup_vault.name

    schedule          = "cron(0 12 * * ? *)"

  }

}
```

2. **Apply Terraform**

Initialize and apply the configuration

```
terraform init

terraform apply
```

    ○

3. **Test and validate recovery**

Azure Backup and Recovery

1. **Set up environment**
    - Install Azure CLI and Terraform.

Authenticate with Azure

```
az login
```

Create a resource group

```
resource "azurerm_resource_group" "rg" {

  name     = "backup-recovery-rg"
  location = "East US"
}
```

Set up Azure Recovery Services vault

```
resource "azurerm_recovery_services_vault" "vault" {
  name                = "backupVault"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  sku                 = "Standard"
}
```

Define backup policy and associate with resources

```
resource "azurerm_backup_policy_vm" "policy" {
  name                = "dailyBackupPolicy"
  resource_group_name = azurerm_resource_group.rg.name
  recovery_vault_name = azurerm_recovery_services_vault.vault.name
  retention_daily     = 7
}
```

Apply Terraform

Initialize and apply the configuration

```
terraform init
terraform apply
```

Test backup and recovery

GCP Backup and Recovery

1. **Set up environment**
    - Install gcloud CLI and Terraform.

Authenticate with GCP

```
gcloud auth application-default login
```

Create a storage bucket

```
resource "google_storage_bucket" "backup_bucket" {

  name          = "my-backup-bucket"

  location      = "US"

  force_destroy = false

}
```

Configure GCP Backup (Cloud Storage Transfer)

```
resource "google_storage_transfer_job" "backup_job" {

  description = "Daily backup job"

  schedule {

    start_date {

      year  = 2024
      month = 11
      day   = 1
    }
    schedule_end_date {
      year  = 2025
      month = 11
      day   = 1
    }
  }
  transfer_spec {
    object_conditions {
      max_time_elapsed_since_last_modification = "2592000s"
    }
    gcs_data_source {
      bucket_name = google_storage_bucket.backup_bucket.name
    }
  }
}
```

Apply Terraform

Initialize and apply the configuration

```
terraform init
terraform apply
```

Test and validate recovery

Oracle Cloud Backup and Recovery

1. **Set up environment**
   - Install OCI CLI and Terraform.

Configure OCI CLI

```
oci setup config
```

Create a bucket in Object Storage

```
resource "oci_objectstorage_bucket" "backup_bucket" {
  namespace = var.namespace
  name      = "backup_bucket"
  compartment_id = var.compartment_id
}
```

Configure backup policies

```
resource "oci_objectstorage_object_lifecycle_policy" "backup_policy" {
  bucket = oci_objectstorage_bucket.backup_bucket.name
  rules {
    name  = "daily-rule"
    action = "ARCHIVE"
    object_name_filter {
      prefix = "backup/"
    }
  }
}
```

Apply Terraform

Initialize and apply the configuration

```
terraform init
terraform apply
```

2. **Test and validate recovery**

Tips for All Providers

- Use version control to manage Terraform files.
- Keep sensitive data in a terraform.tfvars or a secure secrets manager.
- Regularly monitor backup schedules and policies for adherence to requirements.
- Automate recovery testing in non-production environments.

Project 2. Automated Backup Infrastructure

Create a Terraform setup for backing up cloud-based databases, file systems, and storage to secure and reliable backup locations like AWS S3 or Azure Blob Storage.

AWS Backup to S3

Prerequisites

- AWS CLI installed and configured.
- Terraform installed.

- IAM user with programmatic access and permissions for S3 and Backup.

Steps

1. **Create S3 Bucket for Backups**
   - Define the bucket name, region, and versioning.

```
resource "aws_s3_bucket" "backup_bucket" {

  bucket      = "my-backup-bucket"

  acl         = "private"

  versioning {

    enabled = true

  }

}
```

2. **Create Backup Plan**
   - Use AWS Backup to automate the backup schedule.

```
resource "aws_backup_plan" "example" {

  name = "example-backup-plan"

  rule {

    rule_name         = "daily-backup"

    target_vault_name = aws_backup_vault.example.name

    schedule          = "cron(0 12 * * ? *)" # Daily at noon

  }

}
```

3. **Define Backup Vault**
    - Set up a secure vault to store backup data.

```
resource "aws_backup_vault" "example" {

  name        = "example-backup-vault"

  kms_key_arn = aws_kms_key.example.arn

}
```

4. **Assign Resources to Backup**
    - Specify resources like RDS databases, EBS volumes, etc.

```
resource "aws_backup_selection" "example" {

  iam_role_arn = aws_iam_role.example.arn

  plan_id      = aws_backup_plan.example.id

  resources    = ["arnawsrdsus-west-2123456789012dbmydatabase"]

}
```

5. **Deploy Infrastructure**

Run Terraform commands.

```
terraform init

terraform plan

terraform apply
```

Azure Backup to Blob Storage

Prerequisites

- Azure CLI installed and authenticated.
- Terraform installed.
- A storage account created or permissions to create one.

Steps

1. **Create a Storage Account**
    - Define an Azure storage account for backups.

```
resource "azurerm_storage_account" "backup" {

  name                     = "backupstorage"

  resource_group_name      = "my-resource-group"

  location                 = "East US"

  account_tier             = "Standard"

  account_replication_type = "LRS"

}
```

2. **Create a Blob Container**
    - Set up a container for backup data.

```
resource "azurerm_storage_container" "backup_container" {

  name                  = "backups"

  storage_account_name  = azurerm_storage_account.backup.name

  container_access_type = "private"

}
```

3. **Set Up a Backup Policy**
    - Automate backup schedules with Azure Recovery Services.

```
resource "azurerm_backup_policy_vm" "example" {

  name                = "example-policy"

  resource_group_name = "my-resource-group"

  recovery_vault_name = azurerm_recovery_services_vault.example.name

  backup {

    frequency = "Daily"

    time      = "1000"

    retention {

      count = 7

    }

  }

}
```

4. **Add Resources to Backup**
    - Include VMs or databases.

```
resource "azurerm_backup_protected_vm" "example" {

  resource_group_name = "my-resource-group"

  recovery_vault_name = azurerm_recovery_services_vault.example.name

  source_vm_id        = azurerm_virtual_machine.example.id

  policy_id           = azurerm_backup_policy_vm.example.id

}
```

5. **Deploy Infrastructure**

Run Terraform commands.

```
terraform init

terraform plan

terraform apply
```

GCP Backup to Google Cloud Storage

Prerequisites

- GCloud CLI installed and configured.
- Terraform installed.
- A Google Cloud project.

Steps

1. **Create a Cloud Storage Bucket**
   - Define a bucket for backups.

```
resource "google_storage_bucket" "backup" {

  name     = "my-backup-bucket"

  location = "US"

  storage_class = "STANDARD"

}
```

2. **Set IAM Permissions**
   - Allow services to write to the bucket.

```
resource "google_storage_bucket_iam_member" "writer" {

  bucket = google_storage_bucket.backup.name
  role   = "roles/storage.objectCreator"

  member = "serviceAccountbackup-service-account@my-project.iam.gserviceaccount.com"

}
```

3. **Automate Backups**
    - Use Cloud Scheduler or Transfer Service for automation.

```
resource "google_cloud_scheduler_job" "backup_job" {

  name             = "daily-backup-job"

  schedule         = "0 12 * * *"

  time_zone        = "UTC"

  http_target {

    uri         = "https//example.com/backup"

    http_method = "POST"

  }

}
```

4. **Deploy Infrastructure**

Run Terraform commands.

```
terraform init

terraform plan

terraform apply
```

Oracle Backup to Object Storage

Prerequisites

- OCI CLI installed and authenticated.
- Terraform installed.
- Oracle Cloud account.

Steps

- **Create an Object Storage Bucket**
    - Define a bucket for backups.

```
resource "oci_objectstorage_bucket" "backup_bucket" {

  namespace = "my-namespace"

  name      = "backup-bucket"

  compartment_id = var.compartment_id

}
```

- **Set Up IAM Policies**
    - Grant permissions to write to the bucket.

```
resource "oci_identity_policy" "backup_policy" {

  name = "backup-policy"

  statements = [

    "Allow group BackupAdmins to manage buckets in compartment my-compartment",

    "Allow service my-service to use buckets in compartment my-compartment"

  ]

}
```

- **Automate Backups**
    - Use Object Lifecycle Management for automation.

```
resource "oci_objectstorage_object_lifecycle_policy" "example" {
  bucket_name = oci_objectstorage_bucket.backup_bucket.name
  namespace   = "my-namespace"
  rules {
    name        = "delete-old-backups"
    action      = "DELETE"
    object_name_filter {
      include_patterns = ["*.backup"]
    }
  }
}
```

- **Deploy Infrastructure**

Run Terraform commands.

```
terraform init
terraform plan
terraform apply
```

Project 3. 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.

1. AWS Disaster Recovery with Terraform

Step 1 Setup Prerequisites

- Create IAM credentials with the necessary permissions.
- Install Terraform and configure AWS CLI with your credentials.

Step 2 Configure Terraform Backend

- Use an S3 bucket for state files and DynamoDB for state locking.

Example backend.tf

```
terraform {

  backend "s3" {

    bucket         = "your-terraform-backend"

    key            = "disaster-recovery/aws/state.tfstate"

    region         = "us-east-1"

    dynamodb_table = "terraform-state-lock"

  }

}
```

Step 3 Define Primary and Secondary Regions

- Define two AWS regions, one as the primary and the other as the failover.

Example

```
variable "primary_region" {

  default = "us-east-1"

}

variable "secondary_region" {

  default = "us-west-2"
}
```

Step 4 Deploy Resources in Primary Region

- Create resources such as EC2, RDS, and Load Balancers in the primary region.

Example

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


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

Step 5 Configure Replication for DR

- Use AWS services like **Route 53**, **RDS Read Replicas**, and **S3 Cross-Region Replication**.

Example S3 replication

```
resource "aws_s3_bucket_replication_configuration" "replication" {
  role = aws_iam_role.s3_replication.arn
  rule {
    id     = "ReplicationRule"
    status = "Enabled"
    destination {
      bucket = "arnawss3destination-bucket"

      storage_class = "STANDARD"

    }

  }

}
```

Step 6 Automate Failover with Route 53

- Configure Route 53 for DNS failover between regions.

Example

```
resource "aws_route53_health_check" "health_check" {

  type          = "HTTP"

  resource_path = "/health"

  fqdn          = "primary.example.com"

}


resource "aws_route53_record" "failover" {

  zone_id = "Z1234567890"

  name    = "example.com"

  type    = "A"


  alias {

    name                   = aws_lb.primary.dns_name

    zone_id                = aws_lb.primary.zone_id
    evaluate_target_health = true

  }

}
```

---

2. Azure Disaster Recovery with Terraform

Step 1 Setup Prerequisites

- Create a service principal for Terraform access.
- Configure Terraform with the Azure provider.

Step 2 Define Regions and Resource Groups

Example

```
variable "primary_region" {

  default = "East US"

}

variable "secondary_region" {

  default = "West US"

}
```

Step 3 Deploy Resources in Primary Region

Example for a virtual machine

```
resource "azurerm_resource_group" "primary" {

  name     = "primary-rg"

  location = var.primary_region

}
```

```
resource "azurerm_virtual_machine" "primary" {
  name                  = "primary-vm"
  resource_group_name   = azurerm_resource_group.primary.name
  location              = var.primary_region
  network_interface_ids = [azurerm_network_interface.primary.id]
  vm_size               = "Standard_DS1_v2"
}
```

Step 4 Setup Geo-Redundancy

- Use **Azure Site Recovery** or **geo-redundant storage** for DR.

Example for geo-redundant storage

```
resource "azurerm_storage_account" "primary" {
  name                     = "storageacctprimary"
  resource_group_name      = azurerm_resource_group.primary.name
  location                 = var.primary_region
  account_replication_type = "GRS"
}
```

Step 5 Automate Failover

- Use **Azure Traffic Manager** for failover.

Example

```
resource "azurerm_traffic_manager_profile" "failover" {
  name                = "traffic-manager"
  resource_group_name = azurerm_resource_group.primary.name
  location            = var.primary_region
  traffic_routing_method = "Priority"

  dns_config {
    relative_name = "example"
    ttl           = 30
  }

  monitor_config {
    protocol = "HTTP"
    port     = 80
    path     = "/health"
  }
}
```

---

3. GCP Disaster Recovery with Terraform

Step 1 Setup Prerequisites

- Create a GCP project and service account.

- Configure Terraform with GCP provider.

Step 2 Define Regions

Example

```
variable "primary_region" {
  default = "us-central1"
}
variable "secondary_region" {
  default = "us-west1"
}
```

Step 3 Deploy Primary Resources

Example for a Compute Engine instance

```
resource "google_compute_instance" "primary" {
  name         = "primary-instance"
  machine_type = "e2-medium"
  zone         = "${var.primary_region}-a"

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

Step 4 Enable Cross-Region Replication

- Use **Cloud Storage** or **Cloud SQL** for replication.

Example

```
resource "google_storage_bucket" "primary" {
  name     = "primary-bucket"
  location = var.primary_region
  storage_class = "STANDARD"
}
```

Step 5 Automate Failover

- Configure **Cloud DNS** or **load balancing**.

Example for Cloud DNS

```
resource "google_dns_record_set" "failover" {
  name = "example.com."
  type = "A"
  ttl  = 300
  rrdatas = [
    google_compute_instance.primary.network_interface.0.access_config.0.nat_ip
  ]
}
```

---

4. Oracle Cloud Disaster Recovery with Terraform

Step 1 Setup Prerequisites

- Configure Terraform with OCI provider.

Step 2 Define Regions

Example

```
variable "primary_region" {
  default = "us-ashburn-1"
}
variable "secondary_region" {
  default = "us-phoenix-1"
}
```

Step 3 Deploy Resources in Primary Region

Example for a compute instance

```
resource "oci_core_instance" "primary" {
  availability_domain = "UocmUS-ASHBURN-AD-1"
  compartment_id      = var.compartment_ocid
  shape               = "VM.Standard.E2.1"
  display_name        = "primary-instance"
}
```

Step 4 Configure Multi-Region Replication

- Use **Object Storage** or **Database replication**.

Example

```
resource "oci_objectstorage_replication_policy" "policy" {
  name               = "replication-policy"
  destination_bucket = "secondary-bucket"
  destination_region = var.secondary_region

}
```

Step 5 Automate Failover

- Use **OCI DNS**.

Example

```
resource "oci_dns_rr_set" "failover" {

  zone_name_or_id = "example-zone"

  domain          = "example.com"

  rtype           = "A"

  ttl             = 300

  items {

    rdata = oci_core_instance.primary.private_ip

  }

}
```

Project 4. Provisioning Kafka and Event-Driven Architecture

Use Terraform to deploy and manage an Apache Kafka cluster or a managed version (e.g., AWS MSK, Confluent Cloud) for an event-driven microservices architecture.

Here is a step-by-step guide to provision Apache Kafka using Terraform for an event-driven microservices architecture across AWS, Azure, GCP, and Oracle Cloud. This guide covers Terraform usage for creating Kafka clusters, both self-hosted (Apache Kafka) and managed services (AWS MSK, Confluent Cloud, etc.).

1. Setting up Apache Kafka with Terraform (AWS)

Prerequisites

- AWS account
- Terraform installed
- AWS CLI configured

Steps

Create a Terraform configuration file
Create a directory for the project and navigate into it. Inside, create a main.tf file with the following code to deploy AWS MSK (Managed Streaming for Kafka)

```
provider "aws" {

  region = "us-east-1"

}



resource "aws_msk_cluster" "example" {

  cluster_name = "example-cluster"

  kafka_version = "2.8.0"

  number_of_broker_nodes = 3



  broker_node_group_info {

    instance_type = "kafka.m5.large"

    client_subnet_ids = ["subnet-abcde123", "subnet-bcde2345"]  # Your subnet IDs

    security_groups = ["sg-12345678"]

  }



  encryption_info {
    encryption_at_rest {
      data_volume_kms_key_id = "arnawskmsus-east-1123456789012key/abcd-efgh-ijkl-mnop"
    }
  }
}
```

Initialize Terraform
Run the following command to initialize the Terraform configuration

```
terraform init
```

Plan the deployment
Run the following to check the resources Terraform plans to create

```
terraform plan
```

Apply the configuration

Apply the changes to create the Kafka cluster

```
terraform apply
```

Verify the Kafka cluster in the AWS Console
Navigate to the AWS MSK section to ensure the Kafka cluster is deployed successfully.

2. Setting up Apache Kafka with Terraform (Azure)

Prerequisites

- Azure account
- Terraform installed
- Azure CLI configured

}

Steps

Create a Terraform configuration file
In your project directory, create a main.tf for deploying Kafka using Azure's Event Hubs (a managed alternative)

```
provider "azurerm" {

  features {}

}


resource "azurerm_eventhub_namespace" "example" {

  name                = "example-eventhub-namespace"

  location            = "East US"

  resource_group_name = "example-resource-group"

  sku                 = "Standard"

}


resource "azurerm_eventhub" "example" {

  name                 = "example-eventhub"

  namespace_name       = azurerm_eventhub_namespace.example.name

  resource_group_name  = "example-resource-group"

  partition_count      = 4

  message_retention    = 7

  retention_in_days    = 7

}
```

1.

Initialize Terraform
Initialize the configuration

```
terraform init
```

Plan the deployment
Review the planned resources

```
terraform plan
```

Apply the configuration
Deploy the Event Hub and the Kafka-like resource

```
terraform apply
```

Verify the Event Hub
Go to the Azure portal and confirm that the Event Hub and Namespace are created.

---

3. Setting up Apache Kafka with Terraform (GCP)

Prerequisites

- GCP account
- Terraform installed
- GCP CLI configured

Steps

Create a Terraform configuration file
In your project directory, create a main.tf for deploying Confluent Cloud on GCP (you can also use GCP's Pub/Sub for a similar architecture)

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

  region  = "us-central1"

}


resource "google_pubsub_topic" "example" {

  name = "example-topic"

}


resource "google_pubsub_subscription" "example" {

  name  = "example-subscription"

  topic = google_pubsub_topic.example.name

}
```

Initialize Terraform
Run the command to initialize the configuration

```
terraform init
```

Plan the deployment
Preview the resources that Terraform will create

```
terraform plan
```

Apply the configuration
Execute the Terraform apply command to deploy resources

```
terraform apply
```

Verify the Kafka-like services
Check the GCP Console for Pub/Sub and confirm the resources are created.

4. Setting up Apache Kafka with Terraform (Oracle Cloud)

Prerequisites

- Oracle Cloud account
- Terraform installed
- Oracle CLI configured

Steps

Create a Terraform configuration file
Create a main.tf file in your project directory for deploying Kafka in Oracle Cloud

```
provider "oci" {

  region = "us-phoenix-1"

}



resource "oci_streaming_stream" "example" {

  compartment_id = "ocid1.compartment.oc1..xxxxx"

  name           = "example-stream"

  partition_count = 4

}
```

Initialize Terraform
Initialize the configuration

terraform init

Plan the deployment
Run the following to see the execution plan

terraform plan

Apply the configuration
Deploy the Kafka-like service
terraform apply

1. **Verify the Streaming service**
   Navigate to Oracle Cloud Console and check that the streaming service is created.

Conclusion

With these steps, you can provision Kafka or similar event-driven messaging systems using Terraform across different cloud providers. Each cloud provider has its own service for event-driven architecture (AWS MSK, Azure Event Hubs, GCP Pub/Sub, Oracle Streaming), and Terraform makes it easy to automate and manage these resources.

You can customize the configurations to fit your specific infrastructure requirements, like creating Kafka topics, managing access control, or configuring scaling and retention settings.

Other Project

---

1. **Kafka Cluster Automation with Confluent Cloud and Terraform**

Apache Kafka is a powerful event streaming platform enabling applications to publish and consume event messages. **Confluent Cloud** simplifies Kafka deployment by managing the underlying infrastructure on your preferred cloud provider. By leveraging the **Confluent provider**, you can automate Kafka cluster provisioning using **Terraform**.

This guide will walk you through setting up a Kafka cluster, creating topics and service accounts, and configuring fine-grained role-based access controls (RBAC) using Terraform.

Prerequisites

- Familiarity with Kafka concepts such as clusters, topics, and messages. If new to Kafka, refer to Apache Kafka's Introduction to Event Streaming.
- Terraform installed locally or via **HCP Terraform**, which offers features like remote state management and structured plan outputs.

Requirements

- Terraform v1.2+ installed locally
- HCP Terraform account and organization
- Confluent Cloud account (Sign up for a $400 free credit here)

Kafka Consumer Application Setup

Step 1 Add Dependencies

For **Java (Maven)**

xml

```
<dependency>

    <groupId>org.apache.kafka</groupId>

    <artifactId>kafka-clients</artifactId>

    <version>2.8.0</version>
```

```
</dependency>
```

For **Python**

```
pip install kafka-python
```

Step 2 Configure the Kafka Consumer

Java Example

java

```
Properties properties = new Properties();

properties.put("bootstrap.servers",
"your-kafka-cluster-bootstrap-servers");

properties.put("group.id", "orders-consumer-group");

properties.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");

properties.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");

properties.put("auto.offset.reset", "earliest"); // or "latest"


KafkaConsumer<String, String> consumer = new
KafkaConsumer<>(properties);
```

```
consumer.subscribe(Collections.singletonList("orders"));
```

Python Example

python

```
from kafka import KafkaConsumer


consumer = KafkaConsumer(
    'orders',
    group_id='orders-consumer-group',
    bootstrap_servers='your-kafka-cluster-bootstrap-servers',
    auto_offset_reset='earliest',  # or 'latest'
)
for message in consumer
    print(f"Received message {message.value}")
```

Step 3 Start Consuming Messages

In Java

java

```
while (true) {
    ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(1000));

    for (ConsumerRecord<String, String> record  records) {

        System.out.printf("Consumed record with key %s and value
%s%n", record.key(), record.value());

    }

}
```

In Python

python

```
for message in consumer

    print(f"Consumed message {message.value}")
```

Step 4 Monitor Consumer Lag

Run the following command to check consumer lag

```
kafka-consumer-groups --bootstrap-server
your-kafka-cluster-bootstrap-servers \

    --describe --group orders-consumer-group
```

Step 5 Handle Errors and Logging

Java Example

java

```
catch (Exception e) {

    System.err.println("Error while consuming message " + 
e.getMessage());

}
```

Python Example

python

```
import logging


logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(__name__)


try

    for message in consumer

        logger.info(f"Consumed message {message.value}")

except Exception as e

    logger.error(f"Error consuming message {e}")
```

Step 6 Stop the Consumer Application

In Java

java

```
consumer.close();
```

In Python

python

```
consumer.close()
```

Step 7 Clean Up Kafka Resources (Optional)

To delete the `orders` topic

```
kafka-topics --bootstrap-server
your-kafka-cluster-bootstrap-servers --delete --topic orders
```

Step 8 Monitor Kafka Cluster (Optional)

Use tools like **Prometheus** and **Grafana** to track Kafka broker health and monitor consumer lag in production environments.

Notes

- Ensure you have proper access permissions for **Confluent Cloud** and **Terraform**.
- This guide focuses on essential Kafka consumer setup steps. For production deployments, consider advanced features such as high availability, partition management, and message retries.

---

2. Monitoring Infrastructure with HCP Terraform Health Assessments

HCP Terraform's health assessments proactively monitor managed infrastructure to ensure compliance with the intended configuration throughout its lifecycle. Over time, external changes such as service failures, certificate expirations, or manual modifications can impact infrastructure outside Terraform workflows. While Terraform cannot directly prevent such changes, health assessments facilitate rapid detection and resolution.

Components of Health Assessments

Health assessments consist of two primary checks

- **Drift Detection** Ensures infrastructure settings align with the state file in your workspace.
- **Continuous Validation** Verifies resources adhere to compliance checks defined in your configuration.

In this tutorial, you will enable health assessments, perform an on-demand assessment to identify configuration drift, and explore remediation options.

Note

Health assessments are exclusive to the HCP Terraform Plus Edition. Refer to the HCP Terraform pricing page for details.

Prerequisites

Before starting, ensure the following

- An HCP Terraform organization with the Plus edition.
- An HCP Terraform account with owner permissions and locally authenticated.
- Terraform v1.4+ installed locally.
- An AWS account.
- A variable set in HCP Terraform configured with AWS credentials.

Familiarity with Terraform and HCP Terraform workflows is recommended. If you're new, complete the **Terraform Get Started** and **HCP Terraform Get Started** tutorials first.

Step 1 Clone the Example Repository

Clone the repository containing configurations for AWS networking, an EC2 instance, and a security group

```
git clone
https//github.com/hashicorp-education/learn-terraform-drift-dete
ction.git
```

```
cd learn-terraform-drift-detection
```

Open `main.tf` to review the configuration. It includes a security group restricting SSH ingress to a specific CIDR block, representing a private network.

Step 2 Create Infrastructure

Set your HCP Terraform organization name as an environment variable

```
export TF_CLOUD_ORGANIZATION=<your_organization>
```

Initialize the configuration to create the workspace

```
terraform init
```

Provision the infrastructure

```
terraform apply
```

Respond `yes` to confirm. Upon successful completion, note outputs like instance IDs, subnet details, and VPC ID.

Step 3 Simulate Infrastructure Drift

Introduce drift by manually modifying the `bastion_ssh` security group in the AWS console. Replace the source CIDR `192.80.0.0/16` with `0.0.0.0/0`, then save the changes. This creates a drift from the configuration defined in `main.tf`.

Step 4 Enable Health Assessments

Enable health assessments for your workspace to detect drift

1. Navigate to the workspace.
2. In the **Health** section, click **Settings**.
3. Select **Enable**, then save the settings.

HCP Terraform performs assessments every 24 hours using non-actionable, refresh-only plans. Failed runs pause future assessments until resolved.

Step 5 Trigger an On-Demand Assessment

Detect drift immediately

1. Go to the **Health** section of the workspace.
2. Click **Start health assessment**.

HCP Terraform identifies changes, including the modified CIDR block. Note that some AWS-set default values may also appear in the results.

Step 6 Reconcile Drift

If drift is detected, you can

- **Accept the change** Update the configuration to match the new settings, then apply changes to update the state file.
- **Revert the change** Reapply the configuration to restore the original settings.

To revert the drift

```
terraform apply
```

Respond `yes` to confirm. The outputs confirm that the security group has been restored to its original configuration.

Step 7 Verify Drift Resolution

Trigger another on-demand assessment to confirm no further drift exists. The results should indicate that your infrastructure aligns with the intended configuration.

Conclusion

HCP Terraform health assessments offer a robust mechanism for maintaining infrastructure compliance. By detecting and resolving drift promptly, you ensure reliable operations, reduce risks, and minimize unexpected disruptions.

---

1. Validating Terraform Modules with Tests

Terraform tests allow you to verify a module's configuration without impacting existing state files or resources. These tests operate independently from the usual `plan` or `apply` workflows by creating ephemeral infrastructure and validating assertions against its in-memory state. This ensures module changes cean be safely verified without affecting existing infrastructure.

Overview of Testing in Terraform

This guide covers

- Understanding test syntax
- Using helper modules to validate configurations
- Writing and running tests for a sample S3 bucket module

The example module provisions an S3 bucket for hosting a static website. It uses helper modules to generate random bucket names as inputs. You'll learn to

- Run existing tests and analyze their structure
- Write custom tests and helper modules
- Publish the module to the HCP Terraform private registry
- Use test mocking to reduce unnecessary resource creation

Prerequisites

Ensure the following before starting

- Terraform v1.7+ installed locally
- AWS account with credentials configured for Terraform
- GitHub account
- Access to an HCP Terraform account

Step 1 Setting Up the Example Repository

1. Navigate to the template repository.

Click **Use this template** to create a new repository named `terraform-aws-s3-website-tests`.
Clone the repository

```
git clone
https//github.com/USER/terraform-aws-s3-website-tests.git
```

Navigate into the repository

```
cd terraform-aws-s3-website-tests
```

Step 2 Reviewing the Example Configuration

The repository includes

- **`main.tf`** Defines an S3 bucket and uploads website files.
- **`tests`** Contains test files (`.tftest`) and helper modules (`setup/main.tf`).
- **`www`** Holds website content like `index.html` and `error.html`.

Tests consist of

- **Test files** End with `.tftest` and define test assertions.
- **Helper modules** Optional modules for creating resources or data sources for testing.

Example Helper Module (`tests/setup/main.tf`)
Generates a random bucket name

```
resource "random_pet" "bucket_prefix" {
  length = 4

}


output "bucket_prefix" {

  value = random_pet.bucket_prefix.id

}
```

Example Test Assertions (`tests/website.tftest`)

```
run "setup_tests" {

  module {

    source = "./tests/setup"

  }

}


run "create_bucket" {

  command = apply

  variables {

    bucket_name =
"${run.setup_tests.bucket_prefix}-aws-s3-website-test"

  }
  assert {
    condition     = aws_s3_bucket.s3_bucket.bucket ==
"${run.setup_tests.bucket_prefix}-aws-s3-website-test"
    error_message = "Invalid bucket name"
  }
}
```

Step 3 Running Tests

Initialize the configuration

```
terraform init
```

Run the tests

```
terraform test
```

Terraform will execute the tests, validate assertions, and tear down resources automatically.

Step 4 Adding a Custom Test

Create a new helper module (`tests/final/main.tf`)

```
data "http" "index" {
  url    = var.endpoint

  method = "GET"

}


output "status_code" {

  value = data.http.index.status_code

}
```

Add a new test block to `tests/website.tftest`

```
run "website_is_running" {

  command = plan

  module {

    source = "./tests/final"

  }

  variables {

    endpoint = run.create_bucket.website_endpoint

  }

  assert {

    condition     = data.http.index.status_code == 200

    error_message = "Website responded with HTTP status
${data.http.index.status_code}"
```

```
  }
}
```

Initialize and run the new test

```
terraform init
terraform test
```

Step 5 Publishing the Module

Push the changes

```
git add tests/final
git commit -m "Add website_is_running test"
git push origin main
```

1. Publish the module on HCP Terraform
   - Navigate to HCP Terraform Registry.
   - Select **Publish** > **Module** and choose the repository.
   - Configure
     - **Branch Name** `main`
     - **Module Version** `1.0.0`

Step 6 Leveraging Testing in Terraform

By incorporating tests, you gain confidence in the behavior of your Terraform modules. As your module evolves, tests help validate assumptions and maintain stability for users.

3. Validating Terraform Modules with Tests

Terraform tests allow you to verify a module's configuration without impacting existing state files or resources. These tests operate independently from the usual `plan` or `apply` workflows by creating ephemeral infrastructure and validating assertions against its in-memory state. This ensures module changes can be safely verified without affecting existing infrastructure.

Overview of Testing in Terraform

This guide covers

- Understanding test syntax
- Using helper modules to validate configurations
- Writing and running tests for a sample S3 bucket module

The example module provisions an S3 bucket for hosting a static website. It uses helper modules to generate random bucket names as inputs. You'll learn to

- Run existing tests and analyze their structure
- Write custom tests and helper modules
- Publish the module to the HCP Terraform private registry
- Use test mocking to reduce unnecessary resource creation

Prerequisites

Ensure the following before starting

- Terraform v1.7+ installed locally
- AWS account with credentials configured for Terraform
- GitHub account
- Access to an HCP Terraform account

Step 1 Setting Up the Example Repository

1. Navigate to the template repository.

Click **Use this template** to create a new repository named terraform-aws-s3-website-tests.
Clone the repository

```
git clone https//github.com/USER/terraform-aws-s3-website-tests.git
```

Navigate into the repository

```
cd terraform-aws-s3-website-tests
```

Step 2 Reviewing the Example Configuration

The repository includes

- **main.tf** Defines an S3 bucket and uploads website files.
- **tests** Contains test files (.tftest) and helper modules (setup/main.tf).
- **www** Holds website content like index.html and error.html.

Tests consist of

- **Test files** End with .tftest and define test assertions.
- **Helper modules** Optional modules for creating resources or data sources for testing.

Example Helper Module (tests/setup/main.tf)
Generates a random bucket name

```
resource "random_pet" "bucket_prefix" {

  length = 4

}



output "bucket_prefix" {
  value = random_pet.bucket_prefix.id
}
```

Example Test Assertions (tests/website.tftest)

```
run "setup_tests" {
  module {
    source = "./tests/setup"
  }
}

run "create_bucket" {
  command = apply
  variables {
    bucket_name = "${run.setup_tests.bucket_prefix}-aws-s3-website-test"
  }
  assert {
    condition     = aws_s3_bucket.s3_bucket.bucket ==
"${run.setup_tests.bucket_prefix}-aws-s3-website-test"
    error_message = "Invalid bucket name"
  }
}
```

Step 3 Running Tests

Initialize the configuration

```
terraform init
```

Run the tests

```
terraform test
```

Terraform will execute the tests, validate assertions, and tear down resources automatically.

Step 4 Adding a Custom Test

Create a new helper module (tests/final/main.tf)

```
data "http" "index" {
  url    = var.endpoint
  method = "GET"
}

output "status_code" {
  value = data.http.index.status_code
}
```

Add a new test block to tests/website.tftest

```
run "website_is_running" {
  command = plan
  module {
    source = "./tests/final"
  }
  variables {
    endpoint = run.create_bucket.website_endpoint
  }
  assert {
    condition     = data.http.index.status_code == 200
    error_message = "Website responded with HTTP status ${data.http.index.status_code}"
  }
}
```

Initialize and run the new test

```
terraform init
terraform test
```

Step 5 Publishing the Module

Push the changes

```
git add tests/final
git commit -m "Add website_is_running test"
git push origin main
```

1. Publish the module on HCP Terraform

    - Navigate to HCP Terraform Registry.
    - Select **Publish > Module** and choose the repository.
    - Configure
        - **Branch Name** main
        - **Module Version** 1.0.0

Step 6 Leveraging Testing in Terraform

By incorporating tests, you gain confidence in the behavior of your Terraform modules. As your module evolves, tests help validate assumptions and maintain stability for users.

---

4. Mastering Regular Expressions in Terraform A Guide to Regex, RegexAll, and Replace Functions

Regular expressions (regex) are a powerful tool for text pattern matching and manipulation. This guide explores the use of regex in Terraform, including practical examples of the regex, regexall, and replace functions.

What We'll Cover

- What are Regular Expressions?
- What Regex Does Terraform Use?
- The regex Function in Terraform
    - Example Using the regex Function
- The regexall Function in Terraform
    - Example Using the regexall Function
- The replace String Function in Terraform
    - Example Using the replace Function

What Are Regular Expressions?

Regular expressions are patterns used to search, extract, or validate specific data within strings. Developed by mathematician Stephen Cole Kleene in the 1950s, regex has become essential for

- Text search and manipulation
- Data extraction

- Input validation
- Code search and refactoring

What Regex Does Terraform Use?

Terraform uses Google's RE2 regex engine, known for efficiency and reliability. While it supports many regex features, some advanced features like backreferences are not supported. Key supported features include

- **Character Matching** Literal and special characters
- **Repetition** Specifying occurrences of patterns
- **Alternatives** Matching one of several options
- **Grouping and Capturing** Isolating parts of a match

Quick Reference for Regex Patterns

- [a-z] Matches any lowercase letter
- \d+ Matches one or more digits
- ^abc$ Matches the exact string "abc"
- .* Matches any sequence of characters

The regex Function in Terraform

The regex function matches a specified pattern within a string and returns a list of matches. If no match is found, it returns an empty list.

Syntax

```
regex(pattern, string)
```

- **pattern** The regex pattern to match
- **string** The text string to search

Example 1 Extracting a Domain Name

```
variable "website_url" {
  default = "https//www.example.com"
}
```

```
output "extracted_domain" {
  value = regex("^https?//www\\.([^.]+)\\.com$", var.website_url)
}
```

Extracts "example" from the URL.

Example 2 Validating an Email Address

```
variable "email" {
  default = "example@example.com"
}

locals {
  email_valid = length(regex("^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$", var.email)) > 0
}

output "is_valid_email" {
  value = local.email_valid
}
```

Ensures the email follows a valid format.

The regexall Function in Terraform

The regexall function matches all occurrences of a pattern in a string and returns them as a list.

Syntax

```
regexall(pattern, string)
```

Example 1 Finding All Email Addresses

```
variable "content" {
  default = "alice@example.com, bob@example.org"
}
```

```
output "emails" {
  value = regexall("[\\w._%+-]+@[\\w.-]+\\.[a-zA-Z]{2,}", var.content)
}
```

Matches "alice@example.com" and "bob@example.org".

Example 2 Extracting Tags from Logs

```
variable "log_message" {
  default = "tagenv=prod tagregion=us-west"
}

output "tags" {
  value = regexall("tag([^ ]+)=([^ ]+)", var.log_message)
}
```

Extracts all key-value pairs from the log message.

The replace String Function in Terraform

The replace function substitutes a substring in a string with a new value. Unlike regex, it does not support advanced patterns.

Syntax

```
replace(string, old, new)
```

- **string** The text where replacement occurs
- **old** The substring to replace
- **new** The replacement string

Example 1 Replacing "dev" with "prod"

```
variable "old_name" {
  default = "dev-server"
}

output "new_name" {
  value = replace(var.old_name, "dev", "prod")
}
```

Output "prod-server"

Example 2 Modifying a Greeting Message

```
variable "input_string" {
  default = "Hello, World!"
}

locals {
  updated_string = replace(var.input_string, "Hello", "Hi")
}

output "result" {
  value = local.updated_string
}
```

Output "Hi, World!"

Key Differences Between regex and regexall

- **regex** Returns only the first match or an empty list
- **regexall** Captures all matches as a list

Use regex for single-pattern validation and regexall for multiple matches.

Conclusion

Regular expressions are a versatile tool for string manipulation in Terraform. With functions like regex, regexall, and replace, you can handle complex string operations effectively to meet your infrastructure needs.

---

5. Cloud Migration Project

Cloud migration is the process of transferring applications, data, and other business resources from on-premises or legacy systems to cloud platforms. This project illustrates how to execute migrations using

AWS, **Azure**, **GCP**, and **Oracle Cloud**, leveraging **Terraform** to automate the deployment and management of resources.

Cloud Migration Tools Overview

AWS AWS Migration Hub
Centralized tool for tracking migration progress, assessing applications, planning, and monitoring status.

Azure Azure Migrate
Assists in assessing, planning, and migrating workloads to Azure.

GCP Migrate for Compute Engine
Automates the migration of on-premises VMs to Google Cloud Compute Engine.

Oracle OCI Database/Application Migration
Offers tools for migrating databases and application workloads.

Cloud Migration with Terraform

Preparation

Ensure you have

- Access to AWS, Azure, GCP, and Oracle Cloud.
- Terraform installed.
- Cloud credentials for each provider.

Migration Steps

AWS Migration with Terraform
Set up an application on AWS using Terraform

- **Step 1 Configure AWS Provider**

```
provider "aws" {

  region = "us-west-2"
  access_key = "your_aws_access_key"
  secret_access_key = "your_aws_secret_key"
}
```

- **Step 2 Define the Infrastructure**

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

- **Step 3 Apply Configuration**

```
terraform init
terraform plan
terraform apply
```

Azure Migration with Terraform

- **Step 1 Configure Azure Provider**

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

- **Step 2 Define the Infrastructure**

```
resource "azurerm_virtual_machine" "example" {
  name                = "example-vm"
  location            = "East US"
  resource_group_name = "example-resources"
  vm_size             = "Standard_B1ms"
  os_profile {
    computer_name  = "hostname"
    admin_username = "adminuser"
    admin_password = "password123!"
  }
  os_profile_linux_config {
    disable_password_authentication = false
  }
  storage_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "20.04-LTS"
    version   = "latest"
  }
}
```

- **Step 3 Apply Configuration**

```
terraform init

terraform plan

terraform apply
```

GCP Migration with Terraform

- **Step 1 Configure GCP Provider**

```
provider "google" {

  credentials = file("path/to/credentials.json")

  project     = "your-project-id"

  region      = "us-central1"

}
```

- **Step 2 Define the Infrastructure**

```
resource "google_compute_instance" "example" {

  name         = "example-instance"

  machine_type = "f1-micro"

  zone         = "us-central1-a"

  boot_disk {

    initialize_params {

      image = "projects/debian-cloud/global/images/family/debian-10"

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

- **Step 3 Apply Configuration**

```
terraform init
terraform plan
terraform apply
```

Oracle Cloud Migration with Terraform

- **Step 1 Configure Oracle Provider**

```
provider "oci" {
  tenancy_ocid     = "your_tenancy_ocid"
  user_ocid        = "your_user_ocid"
  fingerprint      = "your_fingerprint"
  private_key_path = "path/to/private-key.pem"
  region           = "us-phoenix-1"
}
```

- **Step 2 Define the Infrastructure**

```
resource "oci_core_instance" "example" {
  availability_domain = "UocmPHX-AD-1"
  compartment_id      = "your_compartment_ocid"
  shape               = "VM.Standard2.1"
  source_details {
    source_type = "image"
    image_id    = "your_image_ocid"
  }
  create_vnic_details {
    subnet_id = "your_subnet_ocid"
  }
  metadata = {
    ssh_authorized_keys = "your_ssh_public_key"
  }
}
```

- **Step 3 Apply Configuration**

```
terraform init
terraform plan
terraform apply
```

Finalize the Migration

- Use tools like **AWS Migration Hub**, **Azure Migrate**, or others to track the migration progress.
- Validate the migrated workloads by testing connectivity, performance, and data integrity.

Clean Up

After completing and validating the migration, delete resources using

```
terraform destroy
```

---

6. Edge Computing Solutions

This project offers a beginner-friendly approach to implementing edge computing solutions across multiple cloud platforms (AWS, Azure, GCP, and Oracle Cloud) using Terraform, an Infrastructure as Code (IaC) tool. It simplifies the process of creating and managing resources, including AWS Wavelength zones, Azure IoT Edge, GCP Edge TPU, and Oracle Edge Services.

Key Features

- Automates the provisioning of cloud edge resources using Terraform.
- Includes detailed instructions for initializing, validating, and applying Terraform configurations.
- Provides reusable templates for edge services, such as IoT devices, virtual machines, and edge zones, across different cloud providers.
- Highlights prerequisites, such as CLI installation and cloud account configurations, to ensure smooth implementation.

Terraform Configurations Summary

1. AWS AWS Wavelength

- Configures an AWS Wavelength Zone and deploys an EC2 instance.

Example Configuration

```
provider "aws" {

  region = "us-west-2"

}



resource "aws_wavelength_zone" "example" {
  name   = "wavelength-zone-example"
  region = "us-west-2"
}


resource "aws_instance" "example" {
  ami            = "ami-0abcdef1234567890" # Replace with a valid AMI
  instance_type   = "t3.medium"
  availability_zone = aws_wavelength_zone.example.name
  tags = {
    Name = "WavelengthExample"
  }
}
```

---

2. Azure Azure IoT Edge

- Sets up an Azure IoT Hub and registers an IoT device.

Example Configuration

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


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


resource "azurerm_iothub" "example" {
  name                = "example-iothub"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  sku                 = "S1"
  retention_days      = 7
  partitions_count    = 4
}


resource "azurerm_iot_device" "example" {
  name                = "example-device"
  iothub_name         = azurerm_iothub.example.name
  resource_group_name = azurerm_resource_group.example.name
}
```

---

3. GCP Edge TPU

- Configures a GCP project and provisions an Edge TPU device.

Example Configuration

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

resource "google_project" "example" {
  name       = "example-project"
  project_id = "example-project-id"
  org_id     = "your-org-id"
}

resource "google_edge_tpu_device" "example" {
  name       = "example-edgetpu-device"
  device_id = "edgetpu-device-id"
  zone       = "us-central1-a"
  project    = google_project.example.project_id
}
```

---

4. Oracle Cloud Oracle Edge Services

- Configures Oracle Edge Services and provisions a virtual machine.

Example Configuration

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

resource "oci_edge_services" "example" {
  display_name   = "edge-service-example"

  compartment_id = "ocid1.compartment.oc1..xxxxxxEXAMPLExxxxxx"

  edge_device_id = "ocid1.edge.device.oc1..xxxxxxEXAMPLExxxxxx"

  service_type   = "VirtualMachine"

}


resource "oci_compute_instance" "example" {

  availability_domain = "UocmPHX-AD-1"

  compartment_id      = "ocid1.compartment.oc1..xxxxxxEXAMPLExxxxxx"

  display_name        = "edge-vm-instance"

  shape               = "VM.Standard2.1"

  subnet_id           = "ocid1.subnet.oc1..xxxxxxEXAMPLExxxxxx"


  source_details {

    source_type = "image"

    image_id    = "ocid1.image.oc1..xxxxxxEXAMPLExxxxxx"

  }

}
```

Final Notes

This project introduces edge computing by utilizing Terraform's declarative capabilities for infrastructure management. It is an excellent resource for beginners aiming to explore edge services and understand their implementation on different cloud platforms. Additional configurations can be tailored based on specific needs for advanced setups.

7. Multi-Cloud and Hybrid Cloud Terraform Projects for Each Cloud Provider

To effectively configure resources for Hybrid and Multi-Cloud solutions, we will create a dedicated Terraform project for each cloud provider. This modular approach simplifies resource management and allows for independent deployments or updates.

Prerequisites

- Terraform installed locally
- Credentials with necessary permissions for AWS, Azure, GCP, and Oracle Cloud
- Environment variables set for each provider's authentication

---

AWS: Hybrid and Multi-Cloud Configuration

Directory: aws-terraform

main.tf

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

# AWS Outposts (Hybrid Cloud)
resource "aws_outposts_outpost" "example" {
  name              = "my-outpost"
  site_id           = "outpost-site-id"
  availability_zone = "us-west-2a"
  description       = "My Outpost"
}

resource "aws_outposts_site" "example" {
  name = "outpost-site"
}

# VMware Cloud on AWS (Multi-Cloud)
resource "aws_vpc" "vmware" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_instance" "vmware_instance" {
  ami           = "ami-xxxxxxxx"
  instance_type = "t2.medium"
  subnet_id     = aws_vpc.vmware.id
}
```

Azure: Hybrid and Multi-Cloud Configuration

Directory: azure-terraform

main.tf

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

# Azure Stack (Hybrid Cloud)
resource "azurerm_virtual_network" "example" {
  name                = "example-vnet"
  location            = "East US"
  resource_group_name = "example-resources"
  address_space       = ["10.0.0.0/16"]
}

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

# Azure Arc (Multi-Cloud)
resource "azurerm_kubernetes_cluster" "example" {
  name                = "example-k8s-cluster"
  location            = "East US"
  resource_group_name = "example-resources"
  kubernetes_version  = "1.21.0"
}
```

Google Cloud Platform (GCP): Multi-Cloud Configuration

Directory: gcp-terraform

main.tf

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

# Anthos (Multi-Cloud)
resource "google_container_cluster" "example" {
  name               = "example-cluster"
  location           = "us-central1"
  initial_node_count = 3

  node_config {
    machine_type = "n1-standard-1"
  }
}

resource "google_container_node_pool" "example_pool" {
  name     = "example-node-pool"
  cluster = google_container_cluster.example.name
  location = google_container_cluster.example.location
  node_count = 3

  node_config {
    machine_type = "n1-standard-1"
  }
}
```

Oracle Cloud: Hybrid and Multi-Cloud Configuration

Directory: oracle-terraform

main.tf

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

# Oracle VMware Solution
resource "oci_core_virtual_network" "vmware_vcn" {
  cidr_block       = "10.0.0.0/16"
  display_name    = "vmware_vcn"
  compartment_id  = "your-compartment-id"
  dns_label       = "vcn"
}

# Oracle Cloud Interconnect
resource "oci_core_virtual_network" "oracle_interconnect_vcn" {
  cidr_block      = "192.168.0.0/16"
  display_name    = "oracle-interconnect-vcn"
  compartment_id  = "your-compartment-id"
  dns_label       = "vcn"
}

resource "oci_core_ip_sec_connection" "example" {
  display_name    = "interconnect-connection"
  compartment_id  = "your-compartment-id"
  local_cidr      = "192.168.1.0/24"
  remote_cidr     = "10.0.0.0/16"
}
```

Execution Steps

Initialize each project:

```
cd <project-directory>
terraform init
```

Plan and apply changes:

```
terraform plan
terraform apply
```

Verify resources on each cloud provider's console.

Benefits of Separate Projects

- Isolates configurations per cloud provider
- Easier troubleshooting and debugging
- Simplifies integration with CI/CD pipelines for specific providers

These projects can be customized further for advanced networking, security policies, or specific workloads based on requirements.

---

8. IoT (Internet of Things)

These projects focus on provisioning IoT services across various cloud providers, including necessary configurations. The objective is to create an IoT device registry, connect a device, and send data from the device to the cloud.

AWS IoT Core with Terraform

Pre-requisites

- AWS CLI configured
- Terraform installed with AWS credentials set

Steps

- Create a main.tf file with the following content

```
provider "aws" {

  region = "us-east-1"

}


# Create an IoT Thing (Device)

resource "aws_iot_thing" "iot_thing" {

  name = "my-iot-thing"

}
```

```
# Create an IoT Policy
resource "aws_iot_policy" "iot_policy" {
  name        = "my-iot-policy"
  description = "My IoT Policy"
  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action   = "iot*"
        Effect   = "Allow"
        Resource = "*"
      }
    ]
  })
}

# Create an IoT Certificate
resource "aws_iot_certificate" "iot_certificate" {
  active = true
}

# Attach the policy to the certificate
resource "aws_iot_policy_attachment" "iot_policy_attachment" {
  policy_name = aws_iot_policy.iot_policy.name
  target      = aws_iot_certificate.iot_certificate.arn
}


# Attach the certificate to the IoT Thing
resource "aws_iot_thing_principal_attachment" "iot_thing_attachment" {
  thing_name = aws_iot_thing.iot_thing.name
  principal  = aws_iot_certificate.iot_certificate.arn
}
```

- Initialize and apply Terraform

```
terraform init
terraform apply
```

This creates an IoT Thing, policy, certificate, and the necessary attachments.

Azure IoT Hub with Terraform

Pre-requisites

- Azure CLI configured
- Terraform installed with Azure credentials set

Steps

- Create a main.tf file with the following content

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

# Create a Resource Group
resource "azurerm_resource_group" "rg" {
  name     = "my-iot-rg"
  location = "East US"
}

# Create an IoT Hub
resource "azurerm_iot_hub" "iot_hub" {
  name                = "my-iot-hub"
  location            = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name

  sku {
    name     = "S1"
    capacity = 1
  }
}

# Create an IoT Hub Device
resource "azurerm_iot_hub_device" "iot_device" {
  name                = "my-iot-device"

  iot_hub_name        = azurerm_iot_hub.iot_hub.name

  resource_group_name = azurerm_resource_group.rg.name

  primary_key         = "primary_key_here"

  secondary_key       = "secondary_key_here"

}
```

- Initialize and apply Terraform

```
terraform init

terraform apply
```

This creates an IoT Hub, a resource group, and an IoT device.

GCP Cloud IoT Core with Terraform

Pre-requisites

- Google Cloud SDK configured
- Terraform installed with Google Cloud credentials set

Steps

- Create a main.tf file with the following content

```
provider "google" {

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

# Create a Cloud IoT Registry
resource "google_iot_registry" "iot_registry" {
  name    = "my-iot-registry"
  project = "your-project-id"
  region  = "us-central1"

  device {
    id = "my-device-id"
  }
}

# Create a Cloud IoT Device
resource "google_iot_device" "iot_device" {
  name     = "my-iot-device"
  registry = google_iot_registry.iot_registry.id
  project  = "your-project-id"
  region   = "us-central1"
  id       = "my-device-id"

  public_key {
    format = "RSA_X509_PEM"
    key    = file("public-key.pem")
  }
}
```

- Initialize and apply Terraform

```
terraform init
terraform apply
```

This creates a Cloud IoT Registry and a device.

Oracle IoT Cloud with Terraform

Pre-requisites

- Oracle Cloud CLI configured
- Terraform installed with Oracle Cloud credentials set

Steps

- Create a main.tf file with the following content

```
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"
}

# Create an IoT Stream
resource "oci_iot_stream" "iot_stream" {
  compartment_id = "your-compartment-id"
  display_name   = "my-iot-stream"
  stream_id      = "my-iot-stream-id"
}

# Create an IoT Device Group
resource "oci_iot_device_group" "iot_device_group" {
  compartment_id = "your-compartment-id"
  display_name   = "my-device-group"
  stream_id      = oci_iot_stream.iot_stream.id
}

# Create an IoT Device
resource "oci_iot_device" "iot_device" {
  compartment_id  = "your-compartment-id"
  device_group_id = oci_iot_device_group.iot_device_group.id
  display_name    = "my-iot-device"
  description     = "My IoT device"
  device_id       = "my-iot-device-id"
```

```
}
```

- Initialize and apply Terraform

```
terraform init

terraform apply
```

This creates an IoT Stream, Device Group, and Device in Oracle Cloud.

Conclusion

These steps demonstrate how to provision and manage IoT resources, including device registries, devices, and policies, across AWS, Azure, GCP, and Oracle Cloud using Terraform. These configurations can be extended to include advanced features like data streams, device shadows, or integrations with services like AWS Lambda or Azure Functions for further IoT data processing.