Terraform Real-Time Usecases

With Example Codes

Use Case #1: Setting Up a Simple AWS EC2 Instance

Scenario:

You need to deploy a web server using an AWS EC2 instance to host a static website.

Terraform Code:

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

resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "SimpleWebServer"
  }
}
```

Key Considerations:

- **AMI Selection**: Ensure the AMI is suitable for your region and requirements.
- **Instance Type**: `t2.micro` is chosen for cost-effectiveness, but should be reviewed based on workload.

Use Case #2: Configuring VPC in AWS

Scenario:

Create a custom Virtual Private Cloud (VPC) to support a secure, isolated network for your resources.

Terraform Code:

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

resource "aws_vpc" "custom_vpc" {
  cidr_block = "10.0.0.0/16"
  enable_dns_support = true
  enable_dns_hostnames = true

  tags = {
    Name = "CustomVPC"
  }
}
```

Key Considerations:

- **CIDR Block**: Defines the IP address range; ensure it does not overlap with other networks.
- **DNS Support**: Enables internal DNS hostname resolution.

Use Case #3: Environment Split (Dev, Staging, Prod)

Scenario:

Manage multiple environments such as Development, Staging, and Production using Terraform workspaces to prevent configuration drift and ensure resource isolation.

Terraform Code:

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

resource "aws_instance" "app" {
  count = terraform.workspace == "prod" ? 5 : 1

  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = terraform.workspace == "prod" ? "t2.large" : "t2.micro"

  tags = {
    Name = "AppServer-${terraform.workspace}"
  }
```

```
}
```

Key Considerations:

- **Workspaces**: Use Terraform workspaces to manage different states for each environment.
- **Resource Scaling**: Scale resources based on the environment (e.g., more instances in production).

Use Case #4: Remote State Management

Scenario:

Manage Terraform state files in a secure, shared, and reliable environment using AWS S3 as the backend.

Terraform Code:

```
terraform {
  backend "s3" {
    bucket = "my-terraform-state"
    key    = "global/s3/terraform.tfstate"
    region = "us-east-1"
  }
}
```

Key Considerations:

- **Security**: Ensure the S3 bucket is secure and access is controlled.
- **State Locking**: Use DynamoDB for state locking to prevent concurrent state modifications.

Use Case #5: Autoscaling Setup

Scenario:

Set up an auto-scaling group for EC2 instances to automatically adjust capacity to maintain steady, predictable performance at the lowest possible cost.

Terraform Code:

```
resource "aws_autoscaling_group" "app" {
  launch_configuration = aws_launch_configuration.app.id
  min_size             = 1
  max_size             = 10
  desired_capacity     = 2
  vpc_zone_identifier  = ["subnet-123456"]
```

```
  tag {
    key                 = "Name"
    value               = "AppAutoScalingGroup"
    propagate_at_launch = true
  }
}
```

Key Considerations:

- **Scaling Policies**: Define policies based on CPU utilization or other metrics.
- **Subnet Selection**: Ensure instances are launched in the correct subnets for load balancing.

Use Case #6: RDS Deployment

Scenario:

Deploy an AWS RDS instance to support a relational database service for an application, ensuring it is properly configured for security and performance.

Terraform Code:

```
resource "aws_db_instance" "app_db" {
  allocated_storage    = 20
  storage_type         = "gp2"
  engine               = "mysql"
  engine_version       = "5.7"
  instance_class       = "db.t2.micro"
  name                 = "mydb"
  username             = "dbadmin"
  password             = "securepassword"
  parameter_group_name = "default.mysql5.7"
  multi_az             = false
  skip_final_snapshot  = true
}

resource "aws_security_group" "db" {
  name        = "rds_sg"
  description = "Allow inbound traffic"

  ingress {
    from_port   = 3306
    to_port     = 3306
    protocol    = "tcp"
    cidr_blocks = ["10.0.0.0/16"]
  }
}
```

Key Considerations:

- **Security**: Use a security group to restrict access to the database.

- **Backups and Snapshots**: Configure backups and consider setting `multi_az` for high availability.

Use Case #7: Jenkins Integration

Scenario:

Integrate Terraform with Jenkins to automate the deployment process as part of a CI/CD pipeline.

Terraform Code:

```
// Terraform does not directly manage Jenkins jobs, but you can use Jenkins
to trigger Terraform runs.
// Example Jenkinsfile to trigger Terraform:
pipeline {
  agent any

  stages {
    stage('Deploy') {
      steps {
        script {
          sh 'terraform init'
          sh 'terraform apply -auto-approve'
        }
      }
    }
  }
}
```

Key Considerations:

- **Automation**: Automate `terraform init` and `terraform apply` within Jenkins to streamline deployments.
- **Version Control**: Ensure Terraform scripts are version controlled and reviewed before deployment.

Use Case #8: GitLab CI/CD Pipelines

Scenario:

Configure a GitLab CI/CD pipeline to use Terraform for infrastructure provisioning as part of the software deployment lifecycle.

Terraform Code:

```
# .gitlab-ci.yml example for Terraform
stages:
  - validate
```

```yaml
  - deploy

terraform_validate:
  stage: validate
  script:
    - terraform init
    - terraform validate

terraform_deploy:
  stage: deploy
  script:
    - terraform init
    - terraform apply -auto-approve
  only:
    - master
```

Key Considerations:

- **Pipeline Stages**: Use distinct stages for validation and deployment.
- **Branch Restrictions**: Apply changes only from specific branches (e.g., `master`).

Use Case #9: IAM Configurations

Scenario:

Manage AWS IAM policies and roles using Terraform to ensure proper access control for resources.

Terraform Code:

```hcl
resource "aws_iam_role" "app_role" {
  name = "app_role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17",
    Statement = [{
      Action = "sts:AssumeRole",
      Principal = {
        Service = "ec2.amazonaws.com"
      },
      Effect = "Allow",
      Sid = ""
    }]
  })
}

resource "aws_iam_policy" "app_policy" {
  name        = "app_policy"
  description = "A policy that allows administrative actions"

  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [{
      Action = "ec2:*",
      Effect = "Allow",
```

```
      Resource = "*"
    }]
  })
}
```

Key Considerations:

- **Least Privilege**: Grant the minimum necessary permissions to roles and users.
- **Policy Audit**: Regularly review and update IAM policies to adapt to new requirements.

Use Case #10: Compliance as Code

Scenario:

Implement compliance checks within Terraform scripts to ensure infrastructure meets organizational and regulatory standards.

Terraform Code:

```
resource "aws_instance" "compliant_instance" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  key_name      = "deploy_key"

  tags = {
    CostCenter = "1001"
    Compliance = "SOX"
  }
}

resource "aws_instance_compliance_check" "check" {
  instance_id = aws_instance.compliant_instance.id

  required_tags = [
    "CostCenter",
    "Compliance"
  ]
}
```

Key Considerations:

- **Tagging**: Use tags to manage compliance

-related metadata.

- **Automated Checks**: Implement automated compliance checks to audit resources continuously.

Use Case #11: Multi-Cloud Deployment (AWS, Azure, Google Cloud)

Scenario:

Deploy infrastructure across AWS, Azure, and Google Cloud to ensure high availability and reduce vendor lock-in.

Terraform Code:

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

provider "azurerm" {
  features {}
  location = "East US"
}

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

resource "aws_instance" "aws_web" {
  ami           = "ami-123456"
  instance_type = "t3.micro"
}

resource "azurerm_virtual_machine" "azure_vm" {
  name                  = "azureVM"
  vm_size               = "Standard_B1s"
  delete_os_disk_on_termination = true
}

resource "google_compute_instance" "gcp_vm" {
  name         = "gcpVM"
  machine_type = "e2-micro"
}
```

Key Considerations:

- **Provider Configuration**: Configure each cloud provider within Terraform.
- **Consistency**: Maintain similar capabilities across clouds to facilitate operations and management.

Use Case #12: Kubernetes Cluster Setup

Scenario:

Deploy a Kubernetes cluster across different cloud providers using Terraform to manage containerized applications.

Terraform Code:

```
module "eks" {
  source  = "terraform-aws-modules/eks/aws"
  version = "17.24.0"

  cluster_name    = "my-cluster"
  cluster_version = "1.21"
  subnets         = ["subnet-abcde012", "subnet-bcde012a", "subnet-
fghi345a"]
  vpc_id          = "vpc-1234556abcdef"
}

module "aks" {
  source             = "Azure/aks/azurerm"
  kubernetes_version = "1.21"
  resource_group_name = "myResourceGroup"
  dns_prefix         = "myakscluster"
}

module "gke" {
  source             = "terraform-google-modules/kubernetes-engine/google"
  project_id         = "my-project"
  name               = "my-gke-cluster"
  regional           = true
  region             = "us-central1"
}
```

Key Considerations:

- **Modules**: Use community or provider-supported modules for Kubernetes setup.
- **Version Sync**: Keep Kubernetes versions consistent across providers to ensure compatibility.

Use Case #13: Integrating CloudWatch

Scenario:

Set up AWS CloudWatch for monitoring EC2 instances to track performance and health metrics.

Terraform Code:

```
resource "aws_cloudwatch_metric_alarm" "high_cpu" {
  alarm_name          = "high-cpu-utilization"
  comparison_operator = "GreaterThanOrEqualToThreshold"
  evaluation_periods  = "2"
  metric_name         = "CPUUtilization"
  namespace           = "AWS/EC2"
  period              = "300"
  statistic           = "Average"
  threshold           = "80"
  alarm_description   = "This metric monitors ec2 cpu utilization"
```

```
  dimensions = {
    InstanceId = aws_instance.web.id
  }

  actions_enabled = true
  alarm_actions   = [aws_sns_topic.alerts.arn]
}
```

Key Considerations:

- **Metric and Thresholds**: Define specific metrics and thresholds that match your operational requirements.
- **Action Triggers**: Configure actions to notify or automate responses to metric conditions.

Use Case #14: Log Management with CloudTrail

Scenario:

Configure AWS CloudTrail to manage and monitor logs of AWS account activity, enhancing security and compliance.

Terraform Code:

```
resource "aws_cloudtrail" "main" {
  name                          = "my-cloudtrail"
  s3_bucket_name                = aws_s3_bucket.trail_logs.bucket
  include_global_service_events = true
  is_multi_region_trail         = true
  enable_log_file_validation    = true
}

resource "aws_s3_bucket" "trail_logs" {
  bucket = "my-cloudtrail-logs"
  acl    = "private"
}
```

Key Considerations:

- **Log Validation**: Enable log file validation to ensure log integrity and security.
- **Storage**: Use a dedicated S3 bucket for storing logs securely.

Use Case #15: Reserved Instances Management

Scenario:

Automate the purchasing and management of reserved instances in AWS to optimize costs based on usage predictions.

Terraform Code:

```
resource "aws_ec2_reserved_instances_offering" "cheap_instance" {
  instance_type = "t3.micro"
  availability_zone = "us-west-2a"
  instance_count = 1
  duration = 31536000  # 1 year
  offering_type = "Partial Up

front"
}
```

Key Considerations:

- **Cost Analysis**: Perform thorough cost analysis to determine the feasibility of reserved instances.
- **Reservation Strategy**: Choose the appropriate reservation type (e.g., All Upfront, Partial Upfront, No Upfront) based on cash flow and usage.

Use Case #16: Resource Tagging for Cost Allocation

Scenario:

Implement a resource tagging strategy in AWS to manage costs more effectively by tracking resource usage by different departments, projects, or environments.

Terraform Code:

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

  tags = {
    Project     = "Launch Campaign"
    Department  = "Marketing"
    Environment = "Production"
  }
}
```

Key Considerations:

- **Tagging Policy**: Establish a consistent tagging policy to ensure all resources are tagged correctly.
- **Cost Tracking**: Use AWS Cost Explorer to analyze costs based on tags, helping in budgeting and forecasting.

Use Case #17: Elastic Load Balancing (ELB)

Scenario:

Configure AWS Elastic Load Balancing to distribute incoming application traffic across multiple targets, such as EC2 instances, in multiple Availability Zones, increasing the fault tolerance of your applications.

Terraform Code:

```
resource "aws_elb" "web" {
  name               = "web-load-balancer"
  availability_zones = ["us-west-2a", "us-west-2b", "us-west-2c"]

  listener {
    instance_port     = 80
    instance_protocol = "HTTP"
    lb_port           = 80
    lb_protocol       = "HTTP"
  }

  health_check {
    target              = "HTTP:80/"
    interval            = 30
    timeout             = 5
    healthy_threshold   = 2
    unhealthy_threshold = 2
  }
}
```

Key Considerations:

- **High Availability**: Deploy across multiple Availability Zones to ensure high availability.
- **Health Checks**: Configure health checks to monitor the health of instances and route traffic only to healthy instances.

Use Case #18: CDN Configuration with CloudFront

Scenario:

Set up Amazon CloudFront as a content delivery network to deliver content with lower latency and high transfer speeds to users globally.

Terraform Code:

```
resource "aws_cloudfront_distribution" "s3_distribution" {
  origin {
    domain_name = aws_s3_bucket.mybucket.bucket_regional_domain_name
    origin_id   = "S3-myBucket"
  }

  enabled             = true
  is_ipv6_enabled     = true
  comment             = "S3 distribution for website"
  default_root_object = "index.html"
```

```
  default_cache_behavior {
    allowed_methods = ["DELETE", "GET", "HEAD", "OPTIONS", "PATCH", "POST",
"PUT"]
    cached_methods  = ["GET", "HEAD"]
    target_origin_id = "S3-myBucket"

    forwarded_values {
      query_string = false

      cookies {
        forward = "none"
      }
    }

    viewer_protocol_policy = "redirect-to-https"
    min_ttl                = 0
    default_ttl            = 3600
    max_ttl                = 86400
  }

  price_class = "PriceClass_100"
}
```

Key Considerations:

- **Performance Optimization**: Use caching settings to optimize the delivery and performance of your content.
- **Security**: Configure HTTPS redirection to enhance security and trust.

Use Case #19: Multi-Region Deployment for High Availability

Scenario:

Deploy your application infrastructure across multiple regions to ensure high availability and disaster recovery readiness.

Terraform Code:

```
module "aws_network" {
  source     = "./modules/network"
  regions    = ["us-east-1", "eu-west-1"]
  setup_vpc  = true
  setup_peering = false  # Example if VPC peering is not required
}

resource "aws_instance" "app_server" {
  for_each = toset(["us-east-1", "eu-west-1"])

  provider = aws.instances[each.key]
  ami           = "ami-02354e95b39ca8dec"
  instance_type = "t3.large"
  subnet_id     = module.aws_network.vpc_subnet_id[each.key]
}
```

Key Considerations:

- **Data Residency and Latency**: Consider data residency laws and minimize latency by choosing regions close to your users.
- **Disaster Recovery**: Implement strategies like backup and failover to other regions in case of regional service disruptions.

Use Case #20: Disaster Recovery Setup

Scenario:

Establish a robust disaster recovery plan using Terraform by setting up backup policies and failover mechanisms to ensure business continuity.

Terraform Code:

```
resource "aws_route53_health_check" "primary_site" {
  fqdn              = "api.example.com"
  port              = 443
  type

 = "HTTPS"
  resource_path     = "/health"
  failure_threshold = 3
}

resource "aws_route53_record" "failover" {
  zone_id = aws_route53_zone.primary.zone_id
  name    = "api.example.com"
  type    = "A"
  set_identifier = "secondary"

  failover_routing_policy {
    type = "SECONDARY"
  }

  health_check_id = aws_route53_health_check.primary_site.id
}
```

Key Considerations:

- **Automated Failover**: Use DNS failover to automatically reroute traffic in case of an outage.
- **Health Checks**: Regularly perform health checks on your primary and secondary sites to ensure they are functioning properly.

Use Case #21: Automated Snapshot Management

Scenario:

Automate the process of creating snapshots for EC2 volumes in AWS to ensure data durability and quick recovery in case of failure.

Terraform Code:

```
resource "aws_ebs_volume" "example" {
  availability_zone = "us-west-2a"
  size              = 50

  tags = {
    Name = "MyVolume"
  }
}

resource "aws_ebs_snapshot" "example_snapshot" {
  volume_id = aws_ebs_volume.example.id

  tags = {
    Name = "MySnapshot"
  }
}

resource "aws_lambda_function" "snapshot_scheduler" {
  filename         = "snapshot_scheduler.zip"
  function_name    = "snapshotScheduler"
  role             = aws_iam_role.lambda_exec.arn
  handler          = "exports.handler"
  source_code_hash = filebase64sha256("snapshot_scheduler.zip")
  runtime          = "nodejs12.x"

  environment {
    variables = {
      VOLUME_ID = aws_ebs_volume.example.id
    }
  }
}

resource "aws_cloudwatch_event_rule" "snapshot_rule" {
  schedule_expression = "cron(0 20 * * ? *)"
}

resource "aws_cloudwatch_event_target" "snapshot_target" {
  rule      =   aws_cloudwatch_event_rule.snapshot_rule.name
  target_id = "SnapshotLambda"
  arn       = aws_lambda_function.snapshot_scheduler.arn
}
```

Key Considerations:

- **Automation**: Use AWS Lambda and CloudWatch Events to schedule and automate snapshot creation.
- **Disaster Recovery**: Regular snapshots help in quick recovery and data loss prevention.

Use Case #22: Dynamic Scaling with Terraform

Scenario:

Implement dynamic scaling for a web application using AWS Auto Scaling Groups and Terraform to handle varying load patterns efficiently.

Terraform Code:

```
resource "aws_launch_configuration" "app" {
  name          = "app-launch-configuration"
  image_id      = "ami-0abcd1234abcd1234"
  instance_type = "t2.micro"
  key_name      = "mykey"

  lifecycle {
    create_before_destroy = true
  }
}

resource "aws_autoscaling_group" "app" {
  launch_configuration = aws_launch_configuration.app.id
  min_size             = 1
  max_size             = 10
  desired_capacity     = 2
  vpc_zone_identifier  = ["subnet-12345abcde", "subnet-abcde12345"]

  tag {
    key                 = "Name"
    value               = "AppAutoScaling"
    propagate_at_launch = true
  }
}
```

Key Considerations:

- **Load Balancing**: Integrate with Elastic Load Balancing to distribute incoming traffic across instances.
- **Performance Monitoring**: Monitor performance metrics to adjust scaling policies accordingly.

Use Case #23: Securing Infrastructure with Network ACLs and Security Groups

Scenario:

Enhance security for a cloud environment by configuring network ACLs and security groups to control inbound and outbound traffic effectively.

Terraform Code:

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

```
}

resource "aws_network_acl" "main" {
  vpc_id = aws_vpc.main.id

  egress {
    rule_no    = 100
    action     = "allow"
    cidr_block = "0.0.0.0/0"
    from_port  = 0
    to_port    = 0
    protocol   = "-1"
  }

  ingress {
    rule_no    = 100
    action     = "allow"
    cidr_block = "10.0.0.0/16"
    from_port  = 0
    to_port    = 0
    protocol   = "-1"
  }
}

resource "aws_security_group" "web" {
  vpc_id = aws_vpc.main.id

  ingress {
    from_port   = 80
    to_port     = 80
    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"]
  }

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

Key Considerations:

- **Least Privilege**: Apply the principle of least privilege to network access.
- **Regular Reviews**: Regularly review ACLs and security groups to adapt to changing security needs.

Use Case #24: Multi-T

enant Infrastructure Deployment

Scenario:

Deploy infrastructure for multiple tenants using Terraform, ensuring isolation between tenant resources while optimizing resource utilization.

Terraform Code:

```
variable "tenants" {
  type = list(string)
  default = ["tenant1", "tenant2", "tenant3"]
}

module "tenant_infrastructure" {
  source   = "./modules/tenant_infrastructure"
  for_each = toset(var.tenants)

  tenant_id = each.key
}
```

Key Considerations:

- **Resource Isolation**: Ensure logical separation of resources per tenant.
- **Scalability**: Design modules to be reusable and scalable as new tenants are added.

Use Case #25: Compliance Auditing with Custom Terraform Modules

Scenario:

Use Terraform to enforce compliance standards across your infrastructure by developing custom modules that include required security configurations.

Terraform Code:

```
module "compliant_ec2" {
  source   = "./modules/compliant_ec2"
  instance_count = 5
  compliance_tags = {
    ComplianceFramework = "ISO27001"
    AuditDate           = "2023-09-01"
  }
}
```

Key Considerations:

- **Compliance as Code**: Embed compliance requirements directly into infrastructure code.
- **Audit Trails**: Maintain detailed logs and tags for auditing and compliance tracking.

Use Case #26: Serverless Architecture Deployment

Scenario:

Deploy a serverless architecture using AWS Lambda and API Gateway to handle dynamic, event-driven applications with Terraform.

Terraform Code:

```
resource "aws_lambda_function" "api_handler" {
  function_name = "apiHandler"
  role          = aws_iam_role.lambda_exec.arn
  handler       = "handler.handler"
  runtime       = "nodejs14.x"
  filename      = "path/to/your/deployment/package.zip"
}

resource "aws_api_gateway_rest_api" "api" {
  name        = "ExampleAPI"
  description = "API Gateway for handling responses."
}

resource "aws_api_gateway_method" "api_method" {
  rest_api_id   = aws_api_gateway_rest_api.api.id
  resource_id   = aws_api_gateway_rest_api.api.root_resource_id
  http_method   = "GET"
  authorization = "NONE"
}

resource "aws_api_gateway_integration" "lambda_integration" {
  rest_api_id = aws_api_gateway_rest_api.api.id
  resource_id = aws_api_gateway_method.api_method.resource_id
  http_method = aws_api_gateway_method.api_method.http_method
  integration_http_method = "POST"
  type                    = "AWS_PROXY"
  uri                     = aws_lambda_function.api_handler.invoke_arn
}
```

Key Considerations:

- **Scalability**: Serverless architecture can automatically scale with demand without manual intervention.
- **Cost Efficiency**: Pay only for the compute time you consume, reducing the cost of idle resources.

Use Case #27: Container Orchestration with Terraform

Scenario:

Deploy a Docker container orchestration environment using Amazon ECS with Terraform, managing cluster configurations and service deployments.

Terraform Code:

```
resource "aws_ecs_cluster" "app_cluster" {
  name = "app-cluster"
}

resource "aws_ecs_service" "app_service" {
  name            = "app-service"
  cluster         = aws_ecs_cluster.app_cluster.id
  task_definition = aws_ecs_task_definition.app_task.arn
  desired_count   = 3

  load_balancer {
    target_group_arn = aws_lb_target_group.app_lb_tg.arn
    container_name   = "app"
    container_port   = 8080
  }

  deployment_controller {
    type = "ECS"
  }
}

resource "aws_ecs_task_definition" "app_task" {
  family                   = "app-task"
  requires_compatibilities = ["FARGATE"]
  network_mode             = "awsvpc"
  cpu                      = "256"
  memory                   = "512"
  execution_role_arn       = aws_iam_role.ecs_task_execution.arn

  container_definitions = jsonencode([
    {
      name      = "app"
      image     = "myapp:latest"
      cpu       = 256
      memory    = 512
      essential = true
      portMappings = [
        {
          containerPort = 8080
          hostPort      = 8080
        }
      ]
    }
  ])
}
```

Key Considerations:

- **Cluster Management**: Manage the lifecycle and configuration of the ECS cluster with Terraform.
- **Service Scaling**: Define service scaling policies to handle load variations.

Use Case #28: Infrastructure Monitoring with Prometheus and Grafana

Scenario:

Set up a monitoring stack with Prometheus and Grafana on Kubernetes using Terraform, providing insights into the performance and health of your infrastructure.

Terraform Code:

```
module "k8s_monitoring" {
  source = "git::https://github.com/myorg/terraform-k8s-monitoring.git"

  prometheus_replicas = 2
  grafana_replicas    = 1
}
```

Key Considerations:

- **Data Visualization**: Use Grafana for visualizing data from Prometheus in an informative and actionable way.
- **High Availability**: Ensure high availability of monitoring tools to prevent data loss.

Use Case #29: Managing Cloud Agnostic Infrastructure

Scenario:

Deploy and manage infrastructure across multiple cloud providers (AWS, Azure, Google Cloud) using Terraform, allowing for cloud-agnostic deployments.

Terraform Code:

```
provider "aws" {
  region = "us-west-2"
}
provider "azurerm" {
  features {}
}
provider "google" {
  project = "my-project"
  region  = "us-central1"
}

module "cloud_infrastructure" {
  source = "./modules/multi-cloud-setup"

  aws_region   = "us-west-2"
  azure_region = "East US"
  gcp_region   = "us-central1"
}
```

Key Considerations

:

- **Flexibility**: Maintain flexibility by abstracting cloud-specific resources using modules.
- **Cost Optimization**: Monitor and optimize costs across multiple cloud platforms.

Use Case #30: Dynamic DNS Configuration

Scenario:

Automatically update DNS records in response to changes in IP addresses of resources, using Terraform with AWS Route 53 for dynamic DNS management.

Terraform Code:

```
resource "aws_route53_record" "www" {
  zone_id = aws_route53_zone.primary.zone_id
  name    = "www.example.com"
  type    = "A"
  ttl     = "300"

  records = [aws_instance.web.public_ip]
}
```

Key Considerations:

- **Automation**: Automate DNS updates to ensure minimal downtime and address changes.
- **Reliability**: Improve the reliability of DNS responses with accurate and up-to-date records.