Devops Shack

SCALING DEVOPS INFRASTRUCTURE WITH TERRAFORM

BY DEVOPS SHACK

[Click here for DevSecOps & Cloud DevOps Course]

DevOps Shack

Scaling DevOps Infrastructure with Terraform: Real-World Examples

1. Introduction to Terraform and Infrastructure as Code

- What is Infrastructure as Code (IaC)?
- Benefits of IaC in DevOps workflows
- Why Terraform? (Cloud-agnostic, open-source, scalable)
- Key components: Providers, Resources, Modules, and State
- Terraform's role in automation and consistency

2. How Terraform Enables Infrastructure Scaling

- **Modular Infrastructure Design:**
  Reusable modules allow teams to scale components like networks, compute, and storage consistently across environments.
- **Dynamic Configuration:**
  Use of count, for_each, and variables enables flexible, repeatable resource creation and scaling.
- **Environment Management with Workspaces:**
  Separate development, staging, and production environments with isolated state files for safe and scalable deployments.
- **CI/CD Integration:**
  Automate scaling workflows through integration with DevOps pipelines like GitHub Actions or GitLab CI/CD.

3. Example 1: Scaling AWS Infrastructure with Terraform

- **Compute Scaling:**
  - Auto Scaling Groups (ASG) for EC2 with launch templates
  - Elastic Load Balancer integration

Networking and Storage:

- Scalable VPC setup with multiple subnets
- Provisioning and expanding EBS volumes or S3 buckets

- **Managed Services:**
  - Scaling RDS instances
  - Dynamically deploying Lambda functions and API Gateway

4. Example 2: Scaling GCP Infrastructure with Terraform

- **Compute and Kubernetes:**
  - GKE cluster creation and horizontal pod autoscaling
  - Scalable instance templates and managed instance groups
- **Networking and IAM:**
  - VPC peering and firewall rule expansion
  - Managing IAM roles at scale for services and users
- **Storage and Databases:**
  - Scaling Cloud SQL instances
  - Automated bucket lifecycle and storage class management

5. Example 3: Scaling Azure Infrastructure with Terraform

- **Container and Compute Services:**
  - Provisioning and scaling AKS clusters
  - VM scale sets with auto-scaling rules
- **Network and Application Services:**
  - Load balancer configuration for high availability
  - Scaling App Services with autoscale settings
- **Resource Group Management:**
  - Organized, environment-based scaling

1. Introduction to Terraform and Infrastructure as Code

In today's fast-paced DevOps-driven development lifecycle, manual infrastructure provisioning is no longer viable. **Infrastructure as Code (IaC)** has emerged as a foundational practice in modern DevOps, enabling teams to manage and provision computing infrastructure through machine-readable configuration files. Among the leading tools in the IaC ecosystem is **Terraform**, an open-source solution developed by HashiCorp.

1.1 What is Infrastructure as Code (IaC)?

Infrastructure as Code is the practice of defining infrastructure resources—such as virtual machines, networks, storage, databases, and security rules—using code. Instead of manually configuring environments via GUI or CLI, IaC allows teams to write infrastructure definitions in files, store them in version control (like Git), and automate deployments.

Key benefits of IaC:

- **Version Control:** Infrastructure changes are tracked like application code.
- **Repeatability:** Identical environments can be reproduced across dev, test, and prod.
- **Automation:** Manual errors are minimized through scripted deployment.
- **Speed:** Provision infrastructure in minutes, not hours or days.

1.2 Why Terraform?

Terraform has become the go-to tool for DevOps teams adopting IaC, especially when working with **multi-cloud environments**. Its **declarative syntax**, **provider ecosystem**, and **state management** capabilities make it uniquely suited for managing scalable infrastructure.

Core Advantages of Terraform:

- **Cloud Agnostic:** Supports major cloud providers (AWS, Azure, GCP), Kubernetes, and on-premise solutions via providers.
- **Declarative Language (HCL):** Define your desired infrastructure state without scripting low-level implementation steps.

- **Scalable and Modular:** Reusable modules and dynamic constructs make it ideal for scaling complex systems.
- **State Management:** Terraform tracks the real-time status of infrastructure via a state file, enabling change detection and drift correction.
- **Community and Ecosystem:** A rich library of community-supported modules and tools (like Terraform Cloud and Sentinel) enhances productivity.

1.3 Key Concepts in Terraform

To fully leverage Terraform in a DevOps pipeline, it's important to understand its building blocks:

- **Providers:** Plugins that allow Terraform to interact with APIs from AWS, Azure, GCP, Kubernetes, etc.
- **Resources:** The infrastructure components to be managed, such as aws_instance, azurerm_virtual_machine, or google_compute_instance.
- **Modules:** Encapsulated, reusable configurations that standardize infrastructure across teams and projects.
- **Variables and Outputs:** Parameterization of inputs and structured outputs help with scalability and reusability.
- **State File:** A local or remote file that tracks the current state of deployed infrastructure. Essential for Terraform's plan/apply lifecycle.

This introduction lays the foundation for understanding how Terraform empowers DevOps teams to scale infrastructure efficiently. In the next section, we'll explore **how Terraform's features specifically support scalable infrastructure management**.

2. How Terraform Enables Infrastructure Scaling

Scalability is at the heart of modern infrastructure. As demand grows and systems evolve, DevOps teams must ensure their infrastructure can expand seamlessly without manual intervention or downtime. **Terraform** is built with scalability in mind, offering a structured and automated approach to provisioning, modifying, and replicating infrastructure across diverse environments.

2.1 Modular Infrastructure Design

Terraform encourages breaking down infrastructure into **modules**—self-contained, reusable blocks of configuration. This modular approach allows teams to create standardized building blocks for common components (e.g., VPCs, EC2 instances, Kubernetes clusters), making scaling fast, consistent, and maintainable.

Benefits of modularization:

- Promotes **code reuse** across environments and projects
- Reduces duplication and human error
- Enables easy updates across multiple deployments by changing just one module

2.2 Dynamic Configuration for Flexible Scaling

Terraform provides powerful constructs like count, for_each, and dynamic blocks to create and manage multiple resources with minimal configuration effort.

Examples:

- Scale EC2 instances using count = var.instance_count
- Use for_each to create multiple load balancer listeners from a map
- Implement flexible network rules using dynamic blocks

These constructs allow teams to scale horizontally (adding more instances) or vertically (changing instance size) by simply modifying variables and reapplying the configuration.

2.3 Environment Management with Workspaces

Terraform's **workspaces** feature allows separation of configurations for different environments (e.g., dev, staging, prod) while using the same codebase. Each workspace maintains its own state, enabling isolated deployments with environment-specific scaling policies.

Benefits:

- Cleaner environment management
- Isolated scaling experiments in non-production settings
- Simplified CI/CD pipeline integration for multiple environments

2.4 Integration with CI/CD Pipelines

Terraform fits naturally into modern DevOps pipelines, enabling automatic infrastructure scaling through CI/CD platforms like GitHub Actions, GitLab CI, Jenkins, and others.

Pipeline use cases:

- Automatically scale infrastructure upon merging new code to main
- Validate infrastructure changes through terraform plan in pull requests
- Roll out changes across multiple environments using pipelines and workspaces

This tight integration ensures consistent, repeatable scaling across infrastructure lifecycles without manual intervention.

3. Example 1: Scaling AWS Infrastructure with Terraform

AWS offers scalable cloud services, and Terraform makes it easier to provision and manage these resources programmatically. This section covers scaling across compute, networking, and managed services.

Workflow for Scaling AWS Infrastructure Using

Terraform Step 1: Define Core Infrastructure

Components

- **VPC, Subnets, and Networking**
  - Begin by defining a **Virtual Private Cloud (VPC)** (aws_vpc.example) which isolates your cloud environment.
  - Create one or more **subnets** (aws_subnet.example), to distribute resources across availability zones.
  - Provision an **Internet Gateway** (aws_internet_gateway.example) to enable internet access.
  - Define **Route Tables** (aws_route_table.example) and associate them with subnets for routing traffic.
  - These networking resources set up a scalable and secure infrastructure foundation.

Step 2: Launch Configuration and Auto Scaling Group

- **Launch Configuration (aws_launch_configuration.example)**
  - Defines the EC2 instance setup including AMI, instance type, key pair, security groups, and user data.
  - This serves as a blueprint for the Auto Scaling Group (ASG) to launch EC2 instances.
- **Auto Scaling Group (aws_autoscaling_group.example)**
  - Uses the launch configuration to manage a group of EC2 instances.
  - Initial desired capacity (desired_capacity = 2) specifies how many instances to start with.

- Minimum and maximum sizes (min_size = 2, max_size = 5) define boundaries for scaling.
- The ASG ensures instances are automatically replaced if unhealthy, maintaining availability.

Step 3: Attach Load Balancer

- **Elastic Load Balancer (aws_elb.example)**
  - Distributes incoming traffic across all healthy instances in the ASG.
  - Ensures high availability and fault tolerance by routing requests efficiently.

Step 4: Configure Scaling Policies and CloudWatch Alarms

- **Scaling Policies (aws_autoscaling_policy.scale_up and scale_down)**
  - Define rules for scaling out (adding instances) and scaling in (removing instances).
  - For example:
    - Scale up when average CPU utilization exceeds 70%.
    - Scale down when CPU utilization drops below 30%.
- **CloudWatch Alarms (aws_cloudwatch_metric_alarm.cpu_high and cpu_low)**
  - Monitor EC2 instance metrics such as CPU usage.
  - Trigger the scaling policies based on defined thresholds.

Step 5: Apply Terraform Configuration

- Run terraform init to initialize Terraform working directory.
- Run terraform plan to preview infrastructure changes.
- Run terraform apply to provision resources in AWS.

At this point, the VPC, subnets, load balancer, launch configuration, auto scaling group, and scaling policies are all deployed.

Step 6: Autoscaling in Action

- **Monitoring and Metrics**
  - Amazon CloudWatch continuously monitors instance metrics.
- **Scale Out**
  - When CPU usage rises above 70%, the scaling policy triggers, adding instances up to the max limit (5).
- **Scale In**
  - When CPU usage falls below 30%, the scaling policy triggers removal of instances down to the min limit (2).
- **Terraform's Role**
  - Terraform provisions and manages infrastructure but actual scaling events happen automatically based on CloudWatch alarms and ASG policies.

Step 7: Scaling Network and Additional Services

- Expand subnets or add new availability zones by updating subnet configurations and reapplying Terraform.
- Add managed services (e.g., RDS databases, ElastiCache) to support growing application needs, managed by Terraform.

Step 8: Update and Maintain Infrastructure

- Modify scaling thresholds, instance types, or ASG size parameters in Terraform code to optimize performance and cost.
- Use Terraform's state management and version control to safely update infrastructure.

- Continuously monitor system health and adjust policies based on real- world usage.

3.1 Compute Scaling: EC2 with Auto Scaling and Load Balancer

You can create an **Auto Scaling Group (ASG)** that scales EC2 instances based on CPU usage or scheduled events.

☑ **Terraform Code: Auto Scaling EC2 Instances**

```
resource "aws_launch_template" "web_template"
{ name_prefix  = "web-launch-"
image_id          = "ami-0abcd1234efgh5678"
  instance_type = "t3.micro"
}

resource "aws_autoscaling_group" "web_asg"
  { desired_capacity   = 2
max_size              = 5
min_size              = 1
vpc_zone_identifier = ["subnet-xxxxxxx"]

launch_template {
 id   = aws_launch_template.web_template.id
    version = "$Latest"
  }

tag {
 key                 = "Name"
 value               = "web-asg-instance"
```

```
propagate_at_launch = true
  }
}

resource "aws_autoscaling_policy" "scale_out"
 { name                 = "scale-out"
 scaling_adjustment     = 1
 adjustment_type        =
 "ChangeInCapacity" cooldown = 300
 autoscaling_group_name = aws_autoscaling_group.web_asg.name
}
```

3.2 Networking and Storage Scaling

Terraform can scale network components like **VPCs**, **subnets**, and **EBS volumes** by using variables and dynamic configurations.

☑ **Terraform Code: Scalable VPC and Subnets**

```
variable "azs" {
 default = ["us-east-1a", "us-east-1b", "us-east-1c"]
}

resource "aws_subnet" "multi_az"
 { count         = length(var.azs)
 vpc_id          = aws_vpc.main.id
 availability_zone = var.azs[count.index]
 cidr_block      = cidrsubnet(aws_vpc.main.cidr_block, 4, count.index)
}
```

3.3 Managed Services Scaling: RDS and Lambda

Terraform allows scaling of managed services like **Amazon RDS** by modifying instance types or using Aurora Serverless. For **Lambda**, you can manage memory, concurrency, and triggers.

☑ **Terraform Code: Scalable RDS**

Instance resource "aws_db_instance" "prod_db" { allocated_storage = 100 storage_type = "gp3" engine = "mysql" engine_version = "8.0" instance_class = "db.t3.medium" name = "production" username = "admin" password = "securepass" skip_final_snapshot = true publicly_accessible = false }

☑ **Terraform Code: Lambda with Reserved**

Concurrency resource "aws_lambda_function" "process_data" { function_name = "ProcessDataFn" handler = "index.handler" runtime = "nodejs18.x" memory_size = 512 timeout = 10 role = aws_iam_role.lambda_exec.arn filename =

```
  "lambda_function_payload.zip"
}
```

```
resource "aws_lambda_alias" "live" {
  name            = "live"
  function_name    = aws_lambda_function.process_data.function_name
  function_version = "$LATEST"
}

resource "aws_lambda_provisioned_concurrency_config" "scale_lambda" {
  function_name                     =
aws_lambda_function.process_data.function_name
  qualifier                         =
  aws_lambda_alias.live.name
  provisioned_concurrent_executions = 10
}
```

4. Example 2: Scaling GCP Infrastructure with Terraform

Google Cloud Platform (GCP) offers powerful infrastructure services ideal for scalable applications. Terraform simplifies the orchestration of these services, from compute resources to networking and managed databases.

Workflow for Scaling GCP Infrastructure Using Terraform

Step 1: Define Base Infrastructure

- **Project, Network, and Subnets**
  - You begin by defining the **GCP project** and the **VPC network** (google_compute_network.default), which acts as the container for all networking resources.
  - Then create one or more **subnets** (google_compute_subnetwork.default), specifying regions and IP address ranges.
  - This networking layer provides the foundation for scalable compute resources and service communication.

Step 2: Create Managed Instance Group (MIG)

- **Instance Template (google_compute_instance_template.default)**
  - The instance template defines the VM configuration including machine type, disk image, network interfaces, and metadata.
  - This template standardizes the VMs that will be scaled in the managed instance group.
- **Managed Instance Group (google_compute_region_instance_group_manager.default)**
  - Uses the instance template to create a group of VM instances that can scale automatically.
  - You specify the initial number of instances (target_size = 2), the region, and the subnetwork to deploy the instances.

Step 3: Configure Autoscaling

- **Autoscaler Resource (google_compute_autoscaler.default)**

This resource links directly to the managed instance group.

- Autoscaling is based on metrics such as CPU utilization.
- Example policy:
  - Target CPU utilization is set at 60%.
  - Minimum number of instances: 2
  - Maximum number of instances: 5
- Autoscaler automatically adjusts the size of the managed instance group to meet workload demand.

Step 4: Apply Terraform Configuration

- Run terraform init to initialize modules and provider plugins.
- Run terraform plan to preview infrastructure changes.
- Run terraform apply to create the resources on GCP.

At this point, the managed instance group and autoscaler are deployed and ready to respond to load changes.

Step 5: Autoscaling in Operation

- **Monitoring**
  - GCP automatically collects VM metrics such as CPU usage.
- **Scale Out**
  - When CPU usage exceeds the 60% threshold for a specified period, the autoscaler increases the number of instances, up to the maximum limit of 5.
- **Scale In**
  - When CPU usage falls below the target threshold, the autoscaler decreases the number of instances, but not below the minimum limit of 2.
- **Terraform's Role**

- Terraform manages resource provisioning and updates but does not handle runtime scaling decisions—those are handled by GCP's autoscaler service.

Step 6: Expand Network Dynamically

- The subnet can be expanded or new subnets can be added by modifying the Terraform variable holding subnet ranges.
- Terraform provisions new subnets dynamically using the count parameter, supporting network scalability aligned with compute growth.

Step 7: Scale Managed Services

- **Cloud SQL Instance**
  - Terraform provisions a managed SQL instance (google_sql_database_instance.default) with specific machine type and storage.
  - You can adjust the instance tier or storage size to scale the database vertically.
- **Cloud Functions**
  - Terraform deploys a Cloud Function (google_cloudfunctions_function.default) that automatically scales based on incoming requests.
  - The function is connected to event triggers and automatically scales without manual intervention.

Step 8: Update and Maintain Infrastructure

- Modify autoscaling parameters (e.g., CPU target utilization, max instances) in Terraform files as workload demands change.
- Reapply changes using terraform apply for controlled infrastructure updates.

- Maintain state securely using Terraform remote backends to track infrastructure state consistently.

4.1 Compute Scaling: Google Compute Engine with Managed Instance Groups

Terraform can manage **Managed Instance Groups (MIGs)** that automatically adjust the number of instances based on load.

☑ **Terraform Code: MIG with Autoscaler**

```
resource "google_compute_instance_template" "web_template" {
  name         = "web-template"
machine_type  = "e2-medium"
  region       = "us-central1"

disk {
 source_image = "debian-cloud/debian-11"
    auto_delete  = true
 boot         = true
  }

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

 resource "google_compute_region_instance_group_manager" "web_mig"
  { name          = "web-mig"
region              = "us-central1"
```

```
base_instance_name = "web-instance"

  version {
    instance_template = google_compute_instance_template.web_template.id
  }
  target_size = 2
}

resource "google_compute_autoscaler" "web_autoscaler"
  { name  = "web-autoscaler"
  region = "us-central1"
  target = google_compute_region_instance_group_manager.web_mig.id

  autoscaling_policy
    { max_replicas  = 5
    min_replicas    = 1
    cpu_utilization
    { target = 0.6
    }
  }
}
```

4.2 Scalable GCP Networking: VPC and Subnet Creation

Create multiple subnets across regions using Terraform's dynamic capabilities.

☑ **Terraform Code: Dynamic Subnet Creation**

```
variable "regions" {
  default = ["us-central1", "us-west1"]
```

```
}

resource "google_compute_network" "vpc"
 { name = "terraform-vpc"
 auto_create_subnetworks = false
}

resource "google_compute_subnetwork" "regional_subnets"
 { for_each     = toset(var.regions)
 name           = "subnet-${each.key}"
 region         = each.key
 network        = google_compute_network.vpc.id
 ip_cidr_range  = "10.${lookup(var.region_index, each.key)}.0.0/16"
}

variable "region_index"
 { default = {
   "us-central1" = 1
   "us-west1"    = 2
 }
}
```

4.3 Scaling GCP Managed Services: Cloud SQL and Cloud Functions

Terraform can scale managed services like **Cloud SQL** (similar to AWS RDS) and **Cloud Functions** for serverless compute.

☑ **Terraform Code: Cloud SQL with High Availability**

```
resource "google_sql_database_instance" "production" {
```

```
  name             = "prod-sql"
  database_version =
  "MYSQL_8_0" region      = "us-
  central1"

  settings {
   tier = "db-custom-2-7680" # Custom machine type

   availability_type =
   "REGIONAL"
   backup_configuration
   { enabled = true
   }
  }
}
```

☑ **Terraform Code: Cloud Function with Concurrency**

```
resource "google_cloudfunctions2_function" "data_processor"
{ name      = "dataProcessorFn"
  location    = "us-
  central1" build_config {
   runtime    =   "nodejs20"
   entry_point                =
   "handler" source {
    storage_source {
     bucket = "my-source-bucket"
     object = "function-code.zip"
    }
```

```
}
```

```
  }

  service_config
   { available_memory = "512M"
   timeout_seconds  = 60
   max_instance_count = 10
   min_instance_count = 2
  }
}
```

With Terraform, GCP infrastructure can be scaled programmatically with consistency across regions, zones, and services.

5. Example 3: Scaling Azure Infrastructure with Terraform

Microsoft Azure provides a wide range of scalable cloud services. Terraform simplifies provisioning and scaling Azure resources, including virtual machines, networking, and managed services.

Workflow for Scaling Azure Infrastructure Using Terraform Step

1: Define Base Infrastructure

- **Resource Group & Networking**
  - You start by defining a **resource group** (azurerm_resource_group.example) which acts as a container for all Azure resources.
  - Next, create a **Virtual Network** (azurerm_virtual_network.example) and one or more **subnets** (azurerm_subnet.example), which provide network segmentation for your resources.
  - These resources are foundational and required before deploying scalable compute resources.

Step 2: Configure Virtual Machine Scale Set (VMSS)

- **VMSS Definition (azurerm_linux_virtual_machine_scale_set.example)**
  - VMSS lets you deploy and manage a group of identical VMs.
  - In Terraform, you specify the VM image, size (SKU), number of instances (instances = 2), admin credentials, and networking.
  - Initially, you set a base number of VM instances (e.g., 2) to handle the expected minimum workload.

Step 3: Implement Autoscaling Rules

- **Autoscale Settings (azurerm_monitor_autoscale_setting.example)**
  - Autoscale settings link directly to the VMSS.
  - You define rules based on Azure Monitor metrics, commonly CPU utilization.

- For example:

- If average CPU > 70% for 5 minutes → increase VM instances by 1.
- If average CPU < 30% for 5 minutes → decrease VM instances by 1.

- You also set minimum (minimum = "2") and maximum (maximum = "5") instance counts to control scaling boundaries.

Step 4: Apply Terraform Configuration

- Run terraform init to initialize your working directory.
- Run terraform plan to review changes.
- Run terraform apply to provision the resources on Azure.

At this stage, the VM Scale Set and autoscaling policy are deployed and active.

Step 5: Automatic Scaling in Action

- **Monitoring**
  - Azure Monitor continuously tracks CPU usage of VMs in the scale set.
- **Triggering Scale Actions**
  - When CPU usage exceeds 70%, Azure automatically adds VM instances up to the max limit (5).
  - When CPU drops below 30%, instances are removed down to the min limit (2).
- **Terraform's Role**
  - Terraform manages the lifecycle of VMSS and autoscale settings but does **not** dynamically scale during runtime. The scaling itself is handled by Azure Monitor based on the defined policies.

Step 6: Scaling Networking

- If you need more subnets or to expand network address space, Terraform code uses a **count** loop to create multiple subnets dynamically based on the variable subnet_prefixes.
- This flexibility allows scaling the network alongside compute resources as demand grows.

Step 7: Scaling Managed Services

- Terraform provisions Azure SQL Database inside an **elastic pool** (azurerm_sql_elasticpool.example), which enables automatic resource sharing and scaling across multiple databases.
- For serverless compute, Terraform deploys Azure Function Apps on a **dynamic App Service Plan** (sku { tier = "Dynamic" size = "Y1" }), which scales automatically based on incoming events.

Step 8: Maintenance and Updates

- When you want to change scaling parameters (e.g., increase max VM instances), update Terraform variables or resource blocks.
- Run terraform apply again to adjust resources declaratively.
- Terraform ensures infrastructure changes are made safely and consistently.

5.1 Compute Scaling: Azure Virtual Machine Scale Sets (VMSS)

Azure VM Scale Sets enable you to deploy and manage a set of identical, auto-scaling VMs.

☑ **Terraform Code: VM Scale Set with**

Autoscaling resource "azurerm_resource_group"

"example" { name = "example-resources"

location = "East US"

}

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

resource  "azurerm_subnet"  "example"
 { name            = "internal"
 resource_group_name = azurerm_resource_group.example.name
 virtual_network_name                                    =
 azurerm_virtual_network.example.name   address_prefixes       =
 ["10.0.2.0/24"]
}

resource "azurerm_linux_virtual_machine_scale_set" "example"
 { name           = "examplevmss"
 location           = azurerm_resource_group.example.location
 resource_group_name = azurerm_resource_group.example.name
 sku              = "Standard_DS1_v2"
 instances          = 2
 admin_username      = "adminuser"
 admin_password      =
 "P@ssword1234!"
 source_image_reference {
  publisher = "Canonical"
```

offer =

"UbuntuServer"

```
    sku       = "18.04-LTS"
    version   = "latest"
  }

  network_interface {
    name    = "example-
    nic" primary = true
    ip_configuration {
      name                          = "internal"
      subnet_id                      = azurerm_subnet.example.id
      primary                       = true
      private_ip_address_allocation         = "Dynamic"
    }
  }
}

resource "azurerm_monitor_autoscale_setting" "example" {
  name              = "example-autoscale"
  resource_group_name = azurerm_resource_group.example.name
  target_resource_id = azurerm_linux_virtual_machine_scale_set.example.id
  location          = azurerm_resource_group.example.location

  profile {
    name = "autoscale-profile"

    capacity {
```

```
    minimum = "2"
    maximum = "5"
    default = "2"
   }

   rule
    { metric_trigger
    {
     metric_name       = "Percentage
     CPU" metric_namespace  = ""
     metric_resource_id =
azurerm_linux_virtual_machine_scale_set.example.id
     time_grain        = "PT1M"
     statistic         = "Average"
     time_window       = "PT5M"
     time_aggregation  = "Average"
     operator          =
     "GreaterThan" threshold
                       = 70
    }

    scale_action
     { direction =
     "Increase"
     type      =
     "ChangeCount" value
               = "1"
```

```
    cooldown  = "PT5M"
  }
}
```

```
    rule {
      metric_trigger {
        metric_name        = "Percentage CPU"
        metric_namespace   = ""
        metric_resource_id =
azurerm_linux_virtual_machine_scale_set.example.id
        time_grain         = "PT1M"
        statistic          = "Average"
        time_window        = "PT5M"
        time_aggregation   = "Average"
        operator           = "LessThan"
        threshold          = 30
      }

      scale_action {
        direction = "Decrease"
        type      =
        "ChangeCount" value
                   = "1"
        cooldown  = "PT5M"
      }
    }
  }
}
```

5.2 Networking Scaling: Virtual Networks and Subnets

You can scale network infrastructure by dynamically creating subnets or

expanding address spaces.

☑ Terraform Code: Dynamic Subnet Creation

```
variable "subnet_prefixes" {
 default = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
}

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

resource "azurerm_subnet" "example" {
 count            = length(var.subnet_prefixes)
 name             = "subnet-${count.index +
 1}"
 resource_group_name = azurerm_resource_group.example.name
 virtual_network_name                                    =
 azurerm_virtual_network.example.name    address_prefixes     =
 [var.subnet_prefixes[count.index]]
}
```

5.3 Managed Services Scaling: Azure SQL Database and Functions

Terraform allows you to scale Azure SQL Databases and Azure Functions to handle variable workloads.

☑ Terraform Code: Azure SQL Database with Elastic Pool

```
resource "azurerm_sql_elasticpool" "example"
```

```
{ name                = "example-elasticpool"
resource_group_name = azurerm_resource_group.example.name
```

```
  location            = azurerm_resource_group.example.location
  server_name         = azurerm_sql_server.example.name

  sku_name =
  "StandardPool" dtu        =
  50
  database_dtu_min = 10
  database_dtu_max = 20
  storage_mb       = 5120
}

resource "azurerm_sql_database" "example"
  { name            = "example-db"
  resource_group_name = azurerm_resource_group.example.name
  location          = azurerm_resource_group.example.location
  server_name         = azurerm_sql_server.example.name
  elastic_pool_name  = azurerm_sql_elasticpool.example.name
  sku_name          = "Standard"
  max_size_gb         = 5
}
```

☑ **Terraform Code: Azure Function App with Dynamic Scaling**

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

azurerm_storage_account.example.name

```
  storage_account_access_key =
azurerm_storage_account.example.primary_access_key
  version                    = "~4"

  site_config
   { always_on = true
  }

  app_settings =
   { "FUNCTIONS_WORKER_RUNTIME" = "dotnet"
  }
}

resource "azurerm_app_service_plan" "example"
  { name              = "example-plan"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name

  sku {
    tier = "Dynamic"
    size = "Y1"
  }
}
```

6. Summary and Conclusion

6.1 Summary

In this guide, we explored how Terraform serves as a powerful Infrastructure as Code (IaC) tool to automate and manage scalable DevOps infrastructure across major cloud platforms. We began by understanding the core principles of scaling in DevOps and how Terraform's declarative configuration simplifies infrastructure provisioning and scaling.

We then examined three real-world examples demonstrating Terraform's capabilities:

- **AWS Infrastructure Scaling:** Using Auto Scaling Groups, dynamic subnet allocation, and managed services like RDS and Lambda, Terraform automates scalable deployments tailored to demand.
- **GCP Infrastructure Scaling:** Terraform orchestrates Managed Instance Groups with autoscaling policies, multi-region networking, and managed services like Cloud SQL and Cloud Functions for seamless scalability.
- **Azure Infrastructure Scaling:** Leveraging Virtual Machine Scale Sets, dynamic virtual networks, and elastic managed services such as Azure SQL and Function Apps, Terraform provides robust scaling and management solutions.

These examples highlight Terraform's flexibility, consistency, and reusability, enabling organizations to maintain scalable, cost-effective, and resilient infrastructure.

6.2 Conclusion

Terraform fundamentally transforms how DevOps teams manage infrastructure by promoting automation, version control, and repeatability. Its multi-cloud support empowers organizations to scale resources efficiently while minimizing manual intervention and configuration drift.

By using Terraform to define and manage infrastructure, teams can respond rapidly to changing workloads, optimize resource usage, and maintain high availability. The code-centric approach ensures infrastructure is auditable, reproducible, and easy to maintain, which is critical in fast-paced DevOps environments.

As cloud technologies evolve and infrastructure complexity grows, adopting Terraform for scaling infrastructure becomes not only a best practice but a necessity for modern DevOps success.