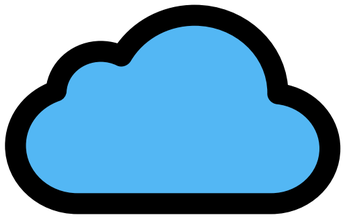
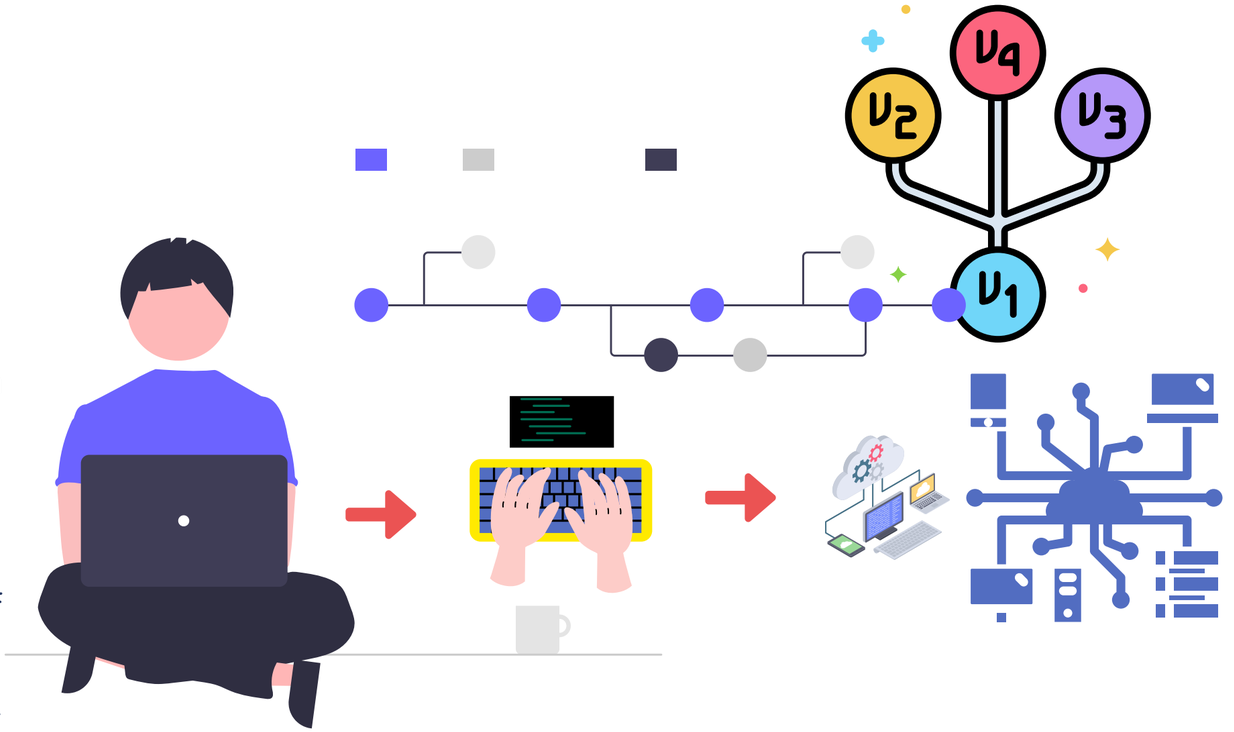


Multi-Cloud Deployment With Terraform

What is Infrastructure as Code?

Infrastructure as Code (IaC) is an approach to managing and provisioning IT infrastructure through code and automation, rather than using manual processes or configuration tools. In this approach, infrastructure is defined and described in code files, which are stored in a version control system, allowing for easy tracking of changes and collaboration among team members.

IaC enables developers and operations teams to automate the process of creating, deploying, and managing infrastructure components, such as servers, networks, and storage, using tools and frameworks designed for this purpose. This results in a more efficient, scalable, and reliable way to manage infrastructure, and it helps ensure consistency and repeatability across different environments (e.g., development, staging, and production).



Popular IaC Tools

1. **Terraform:** An open-source IaC tool created by HashiCorp that supports multiple cloud providers and allows users to create, manage, and modify infrastructure using a declarative language called HCL (HashiCorp Configuration Language).
2. **AWS CloudFormation:** A service provided by Amazon Web Services (AWS) that allows users to define and manage AWS resources using JSON or YAML templates.
3. **Azure Resource Manager (ARM) templates:** A language used for defining and deploying resources in Microsoft Azure cloud infrastructure.
4. **Google Cloud Deployment Manager:** A service provided by Google Cloud Platform (GCP) that enables users to define, deploy, and manage Google Cloud resources using YAML or JSON templates.
5. **Ansible:** An open-source IaC and automation tool that uses a human-readable language called YAML to define infrastructure and tasks for configuration management, application deployment, and other IT processes.
6. **Puppet and Chef:** Both are popular configuration management tools that use a domain-specific language (DSL) to describe and manage infrastructure components.

By adopting IaC, organizations can benefit from improved collaboration, reduced human error, more predictable deployments, and easier management of infrastructure resources.

@Sandip Das

Let's learn more about Terraform

Terraform is an open-source Infrastructure as Code (IaC) tool created by HashiCorp that allows you to provision, manage, and modify infrastructure resources across various cloud providers, on-premises environments, and third-party services. It uses a declarative language called HashiCorp Configuration Language (HCL) to define infrastructure components in a human-readable format.

Terraform enables users to create and manage infrastructure resources by writing configuration files that describe the desired state of the infrastructure. These configuration files are then executed by the Terraform CLI (command-line interface), which interacts with the respective provider APIs to create, update, or delete resources as necessary to match the desired state.

@Sandip Das

Why People Love Terraform?

1. **Provider Support:** Terraform has a wide range of provider plugins, which are integrations with cloud providers, such as AWS, Azure, Google Cloud Platform, and many others. It also supports on-premises providers like VMware and various third-party services, making it versatile and extensible.
2. **Declarative Language:** Terraform uses a declarative language (HCL) that focuses on describing the desired state of the infrastructure, rather than detailing the exact steps to achieve that state. This allows for a more straightforward and maintainable infrastructure definition.
3. **Modularity**: Terraform encourages creating reusable modules that represent common infrastructure patterns, which can be shared and reused across projects, reducing duplication and promoting consistency.
4. **Version Control**: Terraform configuration files can be stored in a version control system, such as Git, enabling collaboration, tracking of changes, and integration with existing development workflows.
5. **State Management**: Terraform keeps track of the current state of your infrastructure in a state file, which allows it to determine the changes required to achieve the desired state. It also supports remote state storage and locking for team collaboration.
6. **Plan & Apply**: Terraform provides a two-step workflow, where it first generates an execution plan (showing the changes to be made) and then applies the changes after manual approval, ensuring greater control and predictability in infrastructure deployments.

Terraform has gained significant popularity among DevOps practitioners and cloud engineers due to its flexibility, extensibility, and ability to manage infrastructure resources in a consistent and scalable manner.

@Sandip Das

Multi Cloud With Terraform!

Multi-cloud with Terraform refers to the practice of using Terraform to manage infrastructure resources across multiple cloud providers (such as AWS, Azure, Google Cloud Platform, etc.) simultaneously within a single project or organization. This approach offers organizations the flexibility to leverage the best services and features from different cloud providers based on their specific requirements and use cases.

Terraform's provider-based architecture makes multi-cloud management possible. Each provider in Terraform is responsible for managing resources within a specific cloud platform or service. By using multiple providers in your Terraform configurations, you can create and manage resources across various cloud platforms, enabling a consistent and unified way of provisioning and managing infrastructure resources in a multi-cloud environment.

Benefits:

1. **Consistency**: Unified infrastructure language across cloud providers.
2. **Simplified Management:** Single IaC tool for multiple platforms.
3. **Flexibility**: Best-of-breed approach for cloud service selection.
4. **Reduced Vendor Lock-in**: Minimize risks with multi-cloud adoption.
5. **Cost Optimization**: Leverage cost-effective services and regional pricing.

@Sandip Das

Configure AWS creds for terraform

There are several ways to configure authentication for Terraform when working with cloud providers like AWS, Azure, or Google Cloud Platform. This answer will focus on AWS, but the general principles apply to other providers as well.

Environment Variables:

export AWS_ACCESS_KEY_ID="your_access_key"
export AWS_SECRET_ACCESS_KEY="your_secret_key"
export AWS_SESSION_TOKEN="your_session_token" # Optional

Shared Credentials File:

Terraform can use the same credentials file as the AWS CLI, which is typically located at ~/.aws/credentials on Linux and macOS or %USERPROFILE%\.aws\credentials on Windows. To set up your credentials file, you can use the aws configure command from the AWS CLI or manually create the file with the following format:

```
[default]
aws_access_key_id = your_access_key
aws_secret_access_key = your_secret_key
```

You can also use named profiles by adding a profitle attribute to the AWS provider block in your Terraform configuration:

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

Static Credentials in Terraform Configuration:

Although not recommended for security reasons, you can provide your access key and secret access key directly in your Terraform configuration using the **access_key** and **secret_key** attributes in the AWS provider block:

```
provider "aws" {
  region     = "us-west-2"
  access_key  = "your_access_key"
  secret_key  = "your_secret_key"
}
```

@Sandip Das

Configure GCP creds for terraform

There are several ways to configure GCP authentication for Terraform:

-

Environment Variable: Set the GOOGLE_APPLICATION_CREDENTIALS environment variable to the path of the JSON key file:

export GOOGLE_APPLICATION_CREDENTIALS="/path/to/your/keyfile.json"

Static Credentials in Terraform Configuration: Although not recommended for security reasons, you can provide the contents of the JSON key file directly in your Terraform configuration using the credentials attribute in the Google provider block:

```
provider "google" {
  project     = "your_project_id"
  region      = "your_region"
  credentials = file("/path/to/your/keyfile.json")
}
```

Application Default Credentials (ADC):
If you don't explicitly set credentials in your Terraform configuration or via the environment variable, Terraform will use ADC, which are automatically provided when running on Google Cloud resources such as Compute Engine, Cloud Run, or Cloud Functions. ADC can also be obtained from the **gcloud** CLI tool with:
gcloud auth application-default login.

Create a service account in your GCP project:

a. Go to the Google Cloud Console.

b. Navigate to "IAM & Admin" > "Service accounts".

c. Click "Create Service Account", provide a name and description, and click "Create".

d. Grant the necessary IAM roles to the service account, based on the resources you want to manage with Terraform, and click "Continue".

e. Click "Done" to finish creating the service account.

Generate a JSON key for the service account:

a. In the list of service accounts, find the one you just created and click the corresponding "Actions" menu (three vertical dots).

b. Select "Manage keys".

c. Click "Add Key" and choose "JSON".

d. A JSON key file will be generated and downloaded to your local machine. Keep this file secure, as it contains sensitive information.

Provide the JSON key file to Terraform:

@Sandip Das

Configure Azure creds for terraform

Azure CLI: If you have the Azure CLI installed and are signed in (az login), Terraform can automatically use your active credentials. No additional configuration is required.

Environment Variables: Set the following environment variables with the appropriate values:
export ARM_CLIENT_ID="your_client_id"
export ARM_CLIENT_SECRET="your_client_secret"
export ARM_SUBSCRIPTION_ID="your_subscription_id"
export ARM_TENANT_ID="your_tenant_id"

Static Credentials in Terraform Configuration: Although not recommended for security reasons, you can provide the credentials directly in your Terraform configuration using the client_id, client_secret, subscription_id, and tenant_id attributes in the AzureRM provider block:

```
provider "azurerm" {
  subscription_id = "your_subscription_id"
  client_id       = "your_client_id"
  client_secret   = "your_client_secret"
  tenant_id       = "your_tenant_id"
}
```

Managed Service Identity (MSI):
When running Terraform on Azure resources like a virtual machine or an Azure Kubernetes Service (AKS) cluster with MSI enabled, Terraform can automatically use the managed identity for authentication. In the AzureRM provider block, set use_msi to true:
hcl

```
provider "azurerm" {
  use_msi = true
}
```

@Sandip Das

Demo Time


Multi-cloud deployment using Terr...
TERRAFORM
WORKSHOP
LIVE
LIVE
Watch on YouTube

main.tf

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

provider "google" {
 project = "mutli-cloud-project"
 region = "us-central1"
 zone = "us-central1-c"
}

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

aws.tf

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

 tags = {
 Name = "example-vpc"
 }
}

resource "aws_internet_gateway" "example" {
 vpc_id = aws_vpc.example.id

 tags = {
 Name = "example-igw"
 }
}

resource "aws_subnet" "example" {
 vpc_id = aws_vpc.example.id
 cidr_block = "10.0.1.0/24"
 map_public_ip_on_launch = true

 tags = {
 Name = "example-subnet"
 }
}

}
```

```
resource "aws_security_group" "example" {
 name = "example"
 description = "Example security group"
 vpc_id = aws_vpc.example.id

 ingress {
 from_port = 22
 to_port = 22
 protocol = "tcp"
 cidr_blocks = ["0.0.0.0/0"]
 }
}

resource "aws_key_pair" "example" {
 key_name = "example-key-pair"
 public_key = file("~/.ssh/id_rsa.pub")
}

resource "aws_instance" "example" {
 ami = "ami-009c5f630e96948cb" # Example Amazon Linux 2023 AMI ID (replace with the appropriate AMI ID for your region)
 instance_type = "t3.small"

 subnet_id = aws_subnet.example.id

 vpc_security_group_ids = [aws_security_group.example.id]

 key_name = aws_key_pair.example.key_name

 tags = {
 Name = "Example Instance"
 }
}
```

@Sandip Das

azure.tf

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

resource "azurerm_virtual_network" "example" {
 name = "example-network"
 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 = ["10.0.1.0/24"]
}

resource "azurerm_network_interface" "example" {
 name = "example-nic"
 location = azurerm_resource_group.example.location
 resource_group_name = azurerm_resource_group.example.name

 ip_configuration {
 name = "internal"
 subnet_id = azurerm_subnet.example.id
 private_ip_address_allocation = "Dynamic"
 public_ip_address_id = azurerm_public_ip.example.id
 }
}
```

```
resource "azurerm_public_ip" "example" {
 name = "example-public-ip"
 location = azurerm_resource_group.example.location
 resource_group_name = azurerm_resource_group.example.name
 allocation_method = "Dynamic"
}

resource "azurerm_linux_virtual_machine" "example" {
 name = "example-vm"
 location = azurerm_resource_group.example.location
 resource_group_name = azurerm_resource_group.example.name
 network_interface_ids = [azurerm_network_interface.example.id]
 size = "Standard_B1s"

 os_disk {
 caching = "ReadWrite"
 storage_account_type = "Standard_LRS"
 }

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

 computer_name = "examplevm"
 admin_username = "adminuser"

 admin_ssh_key {
 username = "adminuser"
 public_key = file("~/.ssh/id_rsa.pub")
 }

 disable_password_authentication = true
}
```

@Sandip Das

gcp.tf

```
resource "google_compute_network" "example" {
 name = "example-network"
 auto_create_subnetworks = false
}

resource "google_compute_subnetwork" "example" {
 name = "example-subnetwork"
 ip_cidr_range = "10.0.0.0/24"
 region = "us-central1"
 network = google_compute_network.example.self_link
}

resource "google_compute_firewall" "example" {
 name = "example-firewall"
 network = google_compute_network.example.self_link

 allow {
 protocol = "tcp"
 ports = ["22"]
 }

 source_ranges = ["0.0.0.0/0"]
}
```

```
resource "google_compute_instance" "example" {
 name = "example-instance"
 machine_type = "f1-micro"

 boot_disk {
 initialize_params {
 image = "projects/debian-cloud/global/images/family/debian-10"
 }
 }

 network_interface {
 subnetwork = google_compute_subnetwork.example.self_link

 access_config {
 // Ephemeral IP
 }
 }

 metadata = {
 ssh-keys = "sandip:${file("~/.ssh/id_rsa.pub")}"
 }
}
```

@Sandip Das

Thank you for joining me on this Multi-cloud Deployment with Terraform Sessopm! I hope you've found it informative and helpful.

I encourage you to continue practicing and experimenting with it.

Thank you again for your time and attention, and I look forward to seeing you in future session!

Follow on: Sandip Das

Subscribe: @LearnTechWithSandip