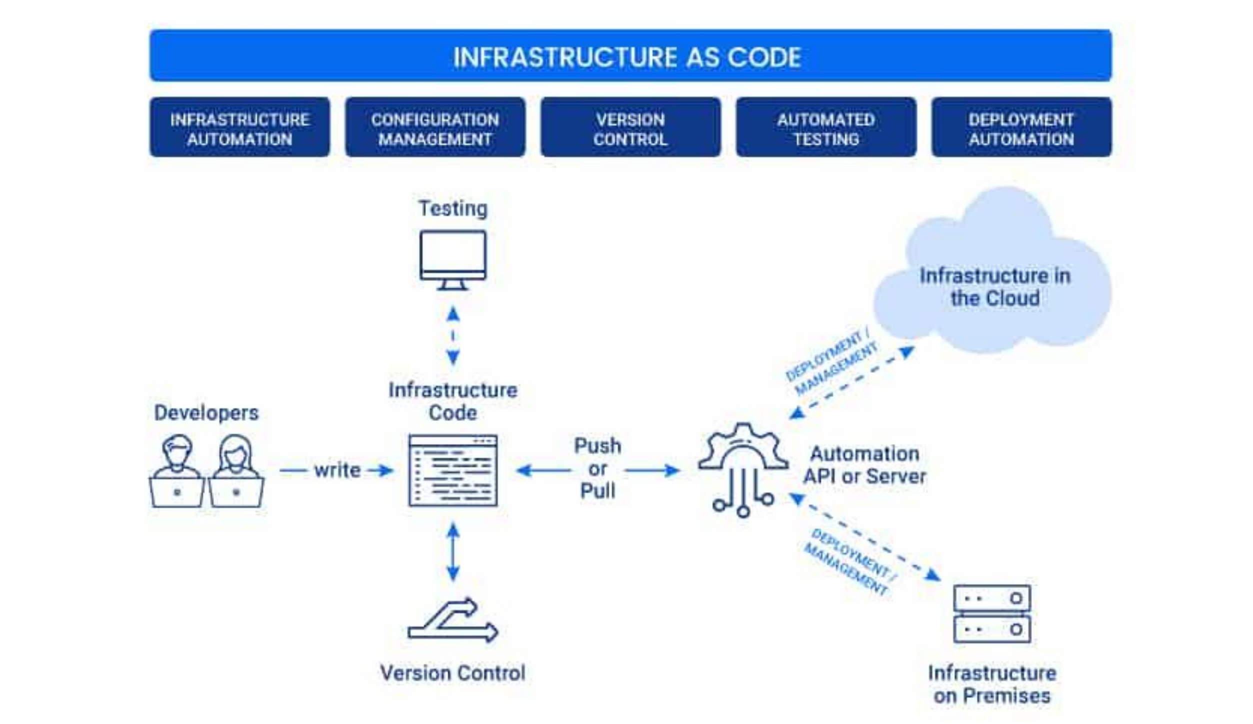
Terraform

What Is Infrastructure as Code (IaC)?

Infrastructure as Code (IaC) is a widespread terminology among DevOps professionals and a key DevOps practice in the industry. It is the process of managing and provisioning the complete IT infrastructure (comprises both physical and virtual machines) using machine-readable definition files. It helps in automating the complete data center by using programming scripts.



Popular IaC Tools:

1. Terraform An open-source declarative tool that offers pre-written modules to build and manage an infrastructure.
2. Chef: A configuration management tool that uses cookbooks and recipes to deploy the desired environment. Best used for Deploying and configuring applications using a pull-based approach.
3. Puppet: Popular tool for configuration management that follows a Client-Server Model. Puppet needs agents to be deployed on the target machines before the puppet can start managing them.
4. Ansible: Ansible is used for building infrastructure as well as deploying and configuring applications on top of them. Best used for Ad hoc analysis.
5. Packer: Unique tool that generates VM images (not running VMs) based on steps you provide. Best used for Baking compute images.
6. Vagrant: Builds VMs using a workflow. Best used for Creating pre-configured developer VMs within VirtualBox.

What Is Terraform?

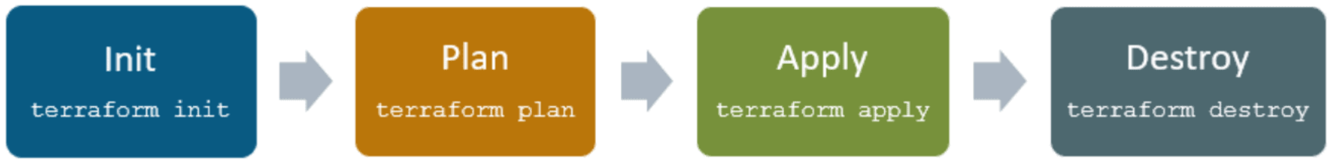
Terraform is one of the most popular **Infrastructure-as-code (IaC) tool**, used by DevOps teams to automate infrastructure tasks. It is used to automate the provisioning of your cloud resources. Terraform is an open-source, cloud-agnostic provisioning tool developed by HashiCorp and written in GO language.

Benefits of using Terraform:

- Does orchestration, not just configuration management
- Supports multiple providers such as AWS, Azure, Oracle, GCP, and many more
- Provide immutable infrastructure where configuration changes smoothly
- Uses easy to understand language, HCL (HashiCorp configuration language)
- Easily portable to any other provider

Terraform Lifecycle

Terraform lifecycle consists of – **init**, **plan**, **apply**, and **destroy**.



1. **Terraform init** initializes the (local) Terraform environment. Usually executed only once per session.
2. **Terraform plan** compares the Terraform state with the as-is state in the cloud, build and display an
execution plan. This does not change the deployment (read-only).
3. **Terraform apply** executes the plan. This potentially changes the deployment.
4. **Terraform destroy** deletes all resources that are governed by this specific terraform environment.

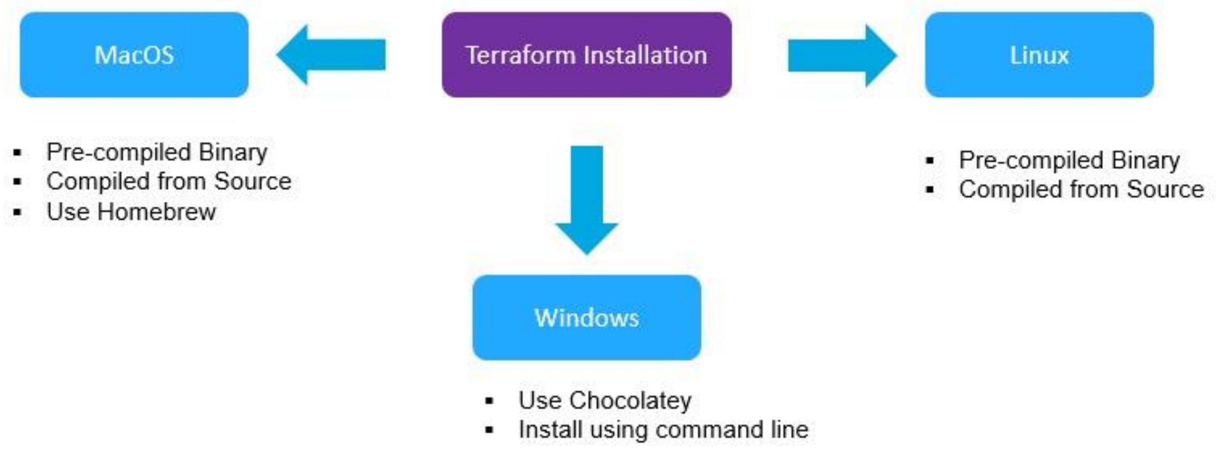
Terraform Core Concepts

1. **Variables**: Terraform has input and output variables, it is a key-value pair. Input variables are used as parameters to input values at run time to customize our deployments. Output variables are return values of a terraform module that can be used by other configurations.
2. **Provider**: Terraform users provision their infrastructure on the major cloud providers such as AWS, Azure, OCI, and others. A *provider* is a plugin that interacts with the various APIs required to create, update, and delete various resources.

3. Module: Any set of Terraform configuration files in a folder is a *module*. Every Terraform configuration has at least one module, known as its ***root module.***
4. State: Terraform records information about what infrastructure is created in a Terraform *state* file. With the state file, Terraform is able to find the resources it created previously, supposed to manage and update them accordingly.
5. Resources: Cloud Providers provides various services in their offerings, they are referenced as Resources in Terraform. Terraform resources can be anything from compute instances, virtual networks to higher-level components such as DNS records. Each resource has its own attributes to define that resource.
6. Data Source: Data source performs a read-only operation. It allows data to be fetched or computed from resources/entities that are not defined or managed by Terraform or the current Terraform configuration.
7. Plan: It is one of the stages in the Terraform lifecycle where it determines what needs to be created, updated, or destroyed to move from the real/current state of the infrastructure to the desired state.
8. Apply: It is one of the stages in the Terraform lifecycle where it applies the changes real/current state of the infrastructure in order to achieve the desired state.

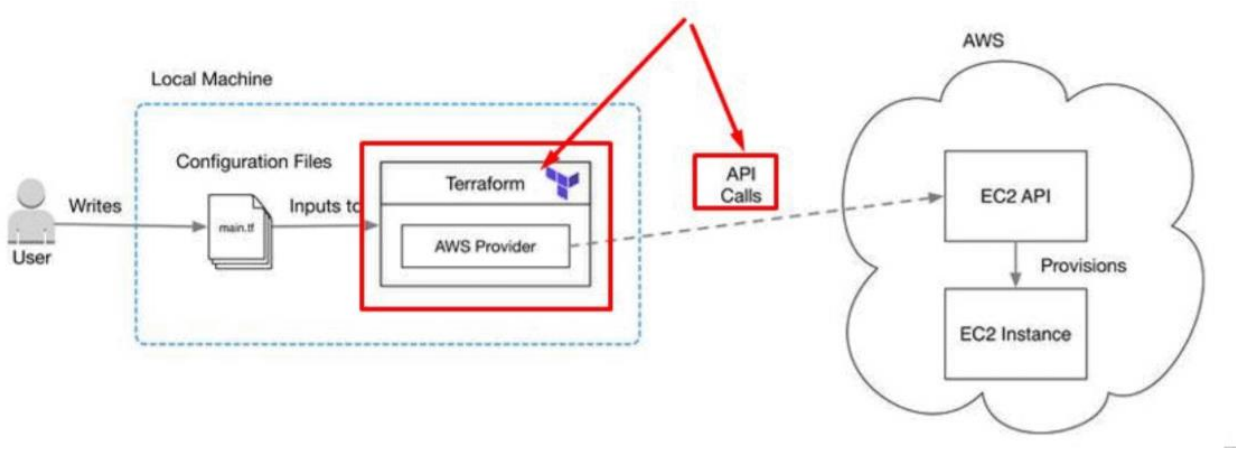
Terraform Installation

Before you start working, make sure you have Terraform installed on your machine, it can be installed on any OS, say Windows, macOS, Linux, or others. Terraform installation is an easy process and can be done in a few minutes.



Terraform Providers

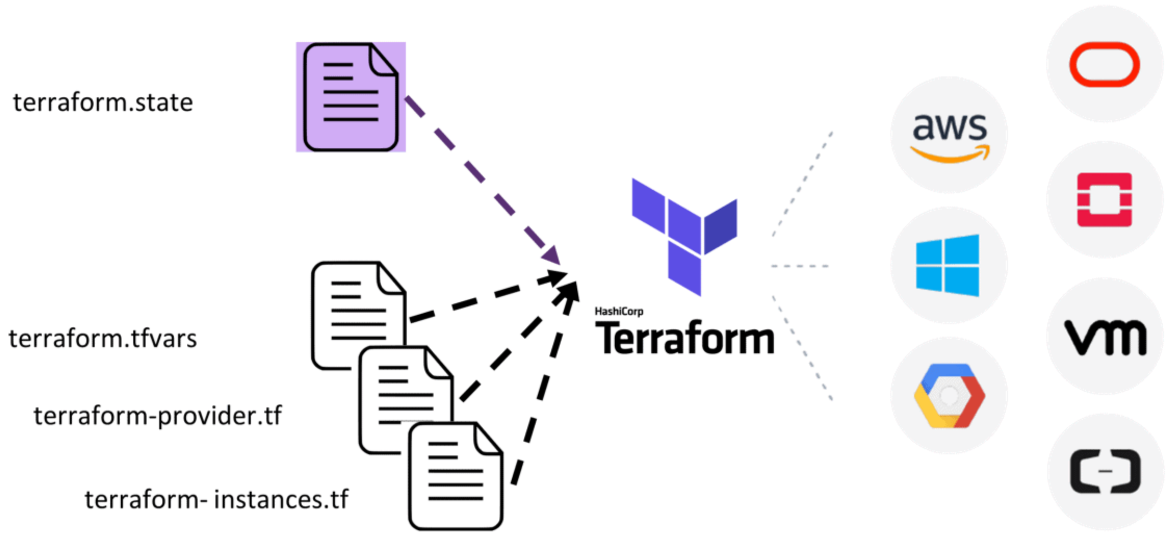
A provider is responsible for understanding API interactions and exposing resources. It is an executable plug-in that contains code necessary to interact with the API of the service. Terraform configurations must declare which providers they require so that Terraform can install and use them.



Terraform has over a hundred providers for different technologies, and each provider then gives terraform user access to its resources. So through AWS provider, for example, you have access to hundreds of AWS resources like EC2 instances, the AWS users, etc.

Terraform Configuration Files

Configuration files are a set of files used to describe infrastructure in Terraform and have the file extensions **.tf** and **.tf.json**. Terraform uses a declarative model for defining infrastructure. Configuration files let you write a configuration that declares your desired state. Configuration files are made up of resources with settings and values representing the desired state of your infrastructure.



A Terraform configuration is made up of one or more files in a directory, provider binaries, plan files, and state files once Terraform has run the configuration.

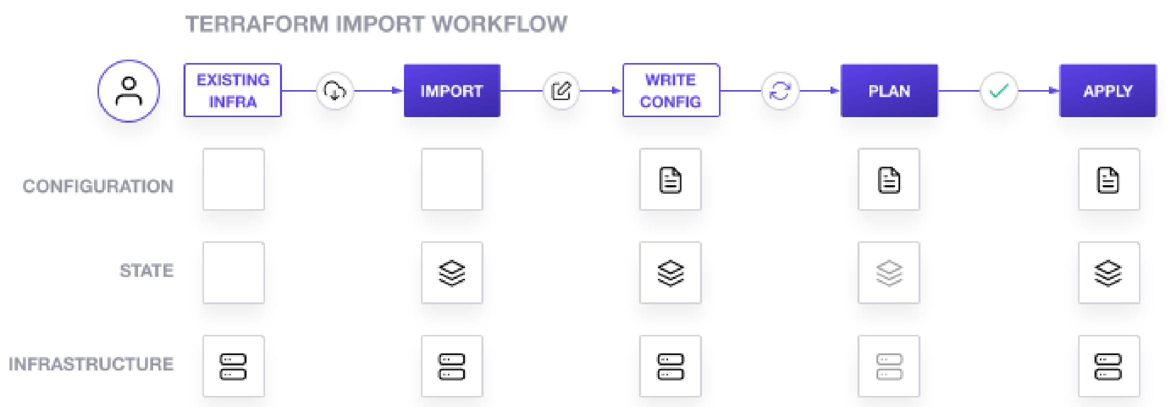
1. Configuration file (*.tf files): Here we declare the provider and resources to be deployed along with the type of resource and all resources specific settings

2. Variable declaration file (variables.tf or variables.tf.json): Here we declare the input variables required to provision resources
3. Variable definition files (terraform.tfvars): Here we assign values to the input variables
4. State file (terraform.tfstate): a state file is created once after Terraform is run. It stores state about our managed infrastructure.

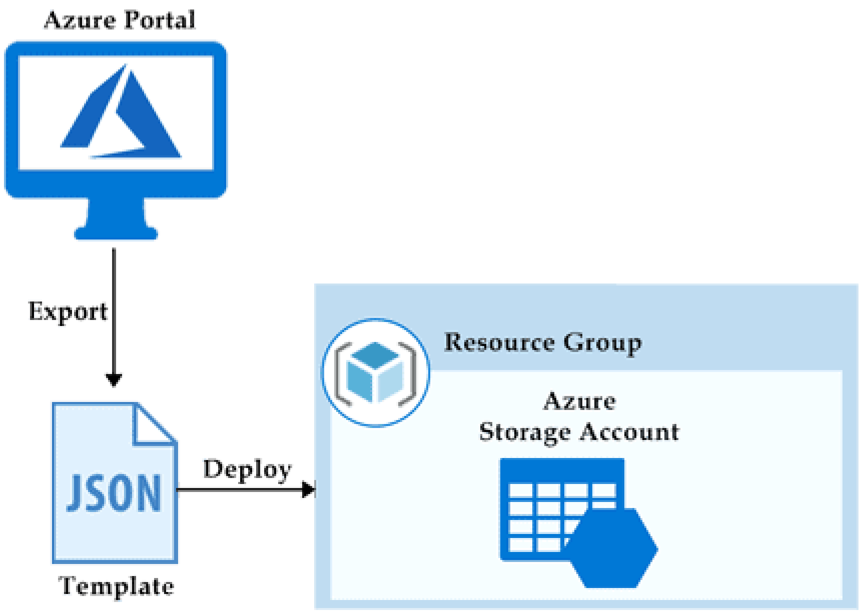
Import Existing Infrastructure

Terraform is one of the great IaC tools with which, you can deploy all your infrastructure's resources. In addition to that, you can manage infrastructures from different cloud providers, such as AWS, Google Cloud, etc. But what if you have already created your infrastructure manually?

Terraform has a really nice feature for importing existing resources, which makes the migration of existing infrastructure into Terraform a lot easier.



What are ARM Template?



ARM (Azure Resource Manager) template is a block of code that defines the infrastructure and configuration for your project. These templates use a declarative syntax to let you define your deployment in the form of JSON (JavaScript Object Notation) files. All the resources and their properties are defined in this template. This helps in automating the deployment process in a constant flow.

Benefits of Using ARM Templates

ARM Templates provides multiple advantages over the deployment process.

- Using ARM Templates, we can declare network infrastructure, storage and any other resources.
- Over the development lifecycle, ARM Templates allows the deployment of resources repeatedly in a consistent manner.
- User can deploy templates parallelly, and only one command is sufficient to deploy all your resource settings.
- Templates can be divided into different modules. In other words, templates can be broken into multiple templates so that a parent template can consist of small templates.
- The PowerShell or Bash Scripts can be added to the templates using deployment scripts.
- The working of ARM Templates can be tested using the ARM template toolkit.
- A user can see the preview of the template. All the resources that are being created or deleted in this template will be shown in the preview.
- A user can integrate templates with Continuous Integration (CI) and Continuous Deployment (CD) tools to automate the release.

Understanding ARM Template

If you are familiar with programming and codes, it will be easier for you to understand the template code. If not, then I will explain you in short and simple terms.

Template Format

The ARM Templates file contains various key-value pairs in the JSON format. For example, below, you can see a format of an ARM Templates.

```
{
"$schema": "https://schema.management.azure.com/schemas/2015-01-
01/deploymentTemplate.json#",
"contentVersion": "1.0.0",
"parameters":{},
"variables":{},
"functions":[],
"resources": [],
"outputs":{}
}
```

- **Schema –** This 'schema' defines the location of the JSON file and specifies the version of the template language that you want to use in this template. This schema depends on the purpose of your deployment. Some schema types are listed below.
    - **Resource Group Deployment** – https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#
    - **Subscription Group Deployment** – https://schema.management.azure.com/schemas/2018-05-01/subscriptionDeploymentTemplate.json#
    - **Management Group Deployment** – https://schema.management.azure.com/schemas/2019-08-01/managementGroupDeploymentTemplate.json#
    - **Tenant Group Deployment** – https://schema.management.azure.com/schemas/2019-08-01/tenantDeploymentTemplate.json#
- **Content Version** – It specifies the version of the templates. This version can be any number that you want to give to your template.

Parameters

Parameters in a templates define the configuration. These parameters are used in run time or during deployment. In a parameter, we need to define the name, type, values and properties. We can also set some allowed values and default values to the parameters, so when a value is not passed during deployment, then the default or allowed values will be used. Below is an example of parameters that defines the type and the default value of username and password for the VM (Virtual Machine).

```
"parameters": {
 "adminUsername": {
  "type": "string",
  "defaultValue": "Admin",
  "metadata": {
   "description": "Username for the Virtual Machine."
    }
  },
  "adminPassword": {
   "type": "securestring",
   "defaultValue": "12345",
   "metadata": {
    "description": "Password for the Virtual Machine."
    }
  }
}
```

Variables

Variables define values used throughout the template. In simple words, you can define a short name for a specific value that can be used anywhere in the template. Variables also become an advantage when you want to update all the values and reference in a template. Then you can update the variable and its value only.

```
"variables": {
 "nicName": "myVMNic",
 "addressPrefix": "10.0.0.0/16",
 "subnetName": "Subnet",
 "subnetPrefix": "10.0.0.0/24",
 "publicIPAddressName": "myPublicIP",
 "virtualNetworkName": "MyVNet"
}
```

Functions

In a template, the function contains the steps and procedures to be followed. It is just like a variable that defines the steps performed when called in a templates. The below example of the function defines the unique name for the resources.

```
"functions": [
 {
  "namespace": "contoso",
  "members": {
   "uniqueName": {
    "parameters": [
      {
       "name": "namePrefix",
       "type": "string"
      }
    ],
    "output": {
     "type": "string",
     "value": "[concat(toLower(parameters('namePrefix')),
uniqueString(resourceGroup().id))]"
         }
       }
     }}],
```

Resources

All the azure resources are defined here that makes the deployment. For creating a resource, we need to set up the type, name, location, version and properties of the resource that needs to be deployed. We can also use the variables and parameters here that are defined in the ‘variables’ section. Below is the example of declaring the resources in a templates.

```
"resources": [
 {
  "type": "Microsoft.Network/publicIPAddresses",
  "name": "[variables('publicIPAddressName')]",
  "location": "[parameters('location')]",
  "apiVersion": "2018-08-01",
  "properties": {
    "publicIPAllocationMethod": "Dynamic",
    "dnsSettings": {
    "domainNameLabel": "[parameters('dnsLabelPrefix')]"
    }
  }
}
],
```

Outputs

Output defines the result that you want to see when a template runs. In simple words, the final words that you want to see when a template is successfully deployed. In the below example, the hostname with a value fetched from the public IP address name.

```
"outputs": {
 "hostname": {
  "type": "string",
  "value":
"[reference(variables('publicIPAddressName')).dnsSettings.fqdn]"
 }
}
```