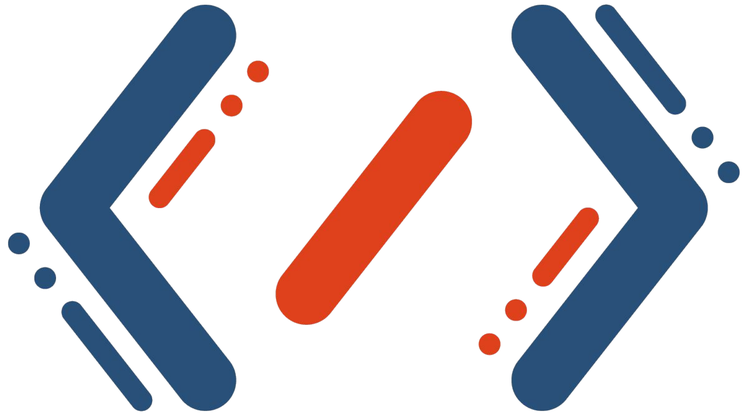


DEVWEEK

CODING THE FUTURE – WFH EDITION

NOVEMBER 23 - 27

MEDELLIN

Terraform Hands-On

DevOps Community - DevWeek Nov/2020

INFRASTRUCTURE CHALLENGES

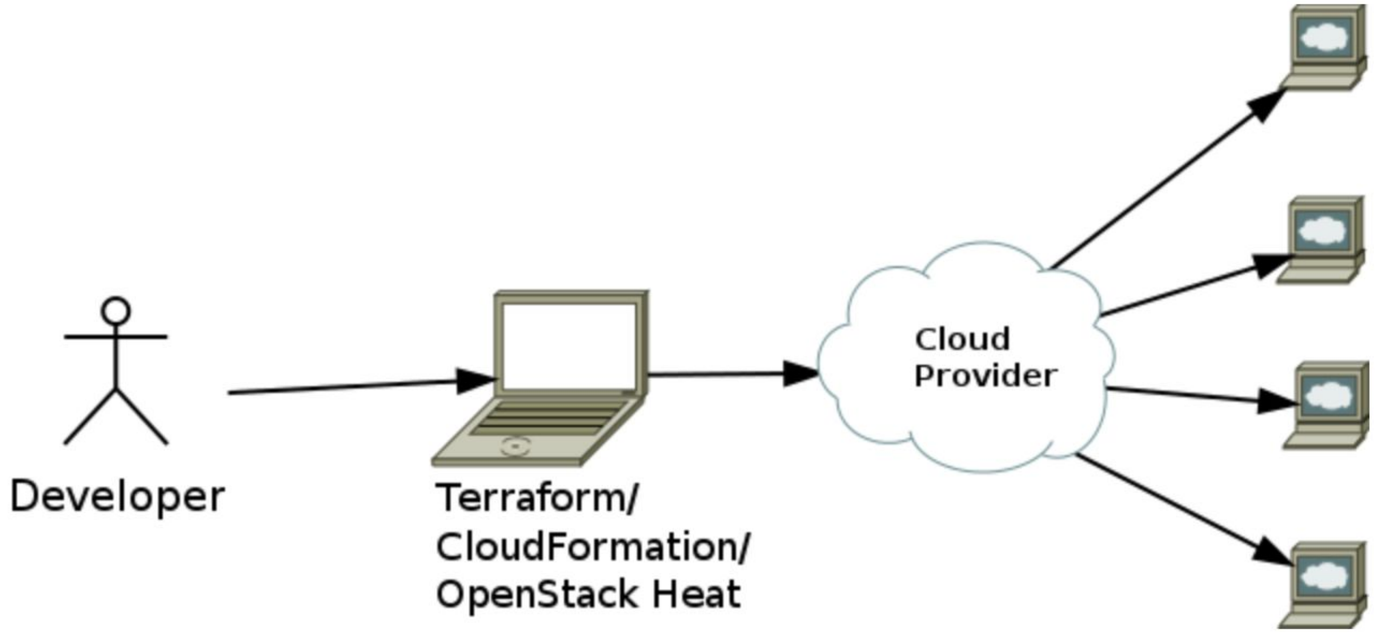
- Need to learn to code
- Don’t know the change impact.
- Need to revert the change
- Can’t track changes
- Can’t automate a resource
- Multiple environments for infrastructure

✓ **Terraform has been created to solve these challenges.**

INFRASTRUCTURE AS CODE TOOLS

Provisioning

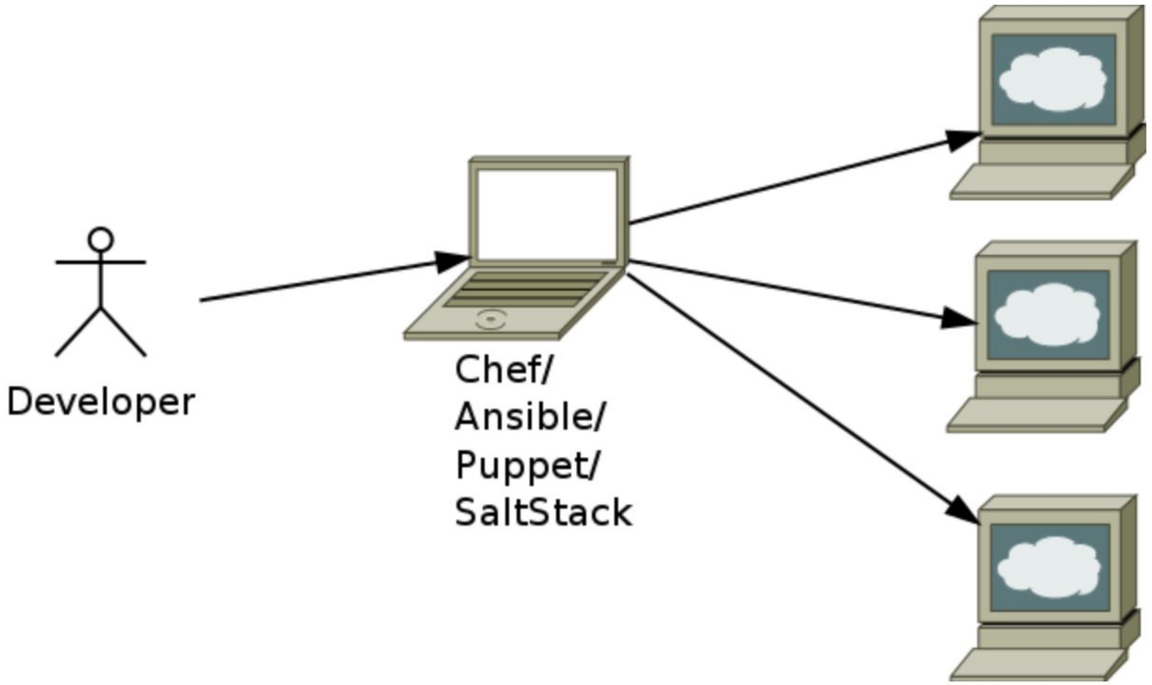
Used to create servers, database servers, load balancers, subnets, firewalls, and all other components of your infrastructure. These tools **make API calls to providers** to create the required infrastructure.



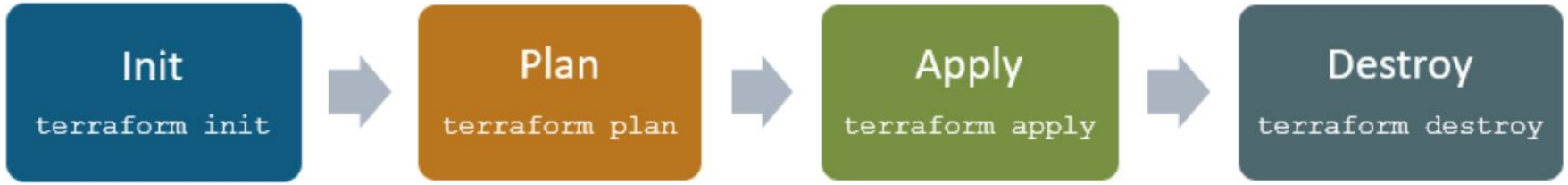
INFRASTRUCTURE AS CODE TOOLS

Configuration management

Designed to manage users, install and manage software and tools **on existing servers.**



WORKFLOW



source: geekflare.com

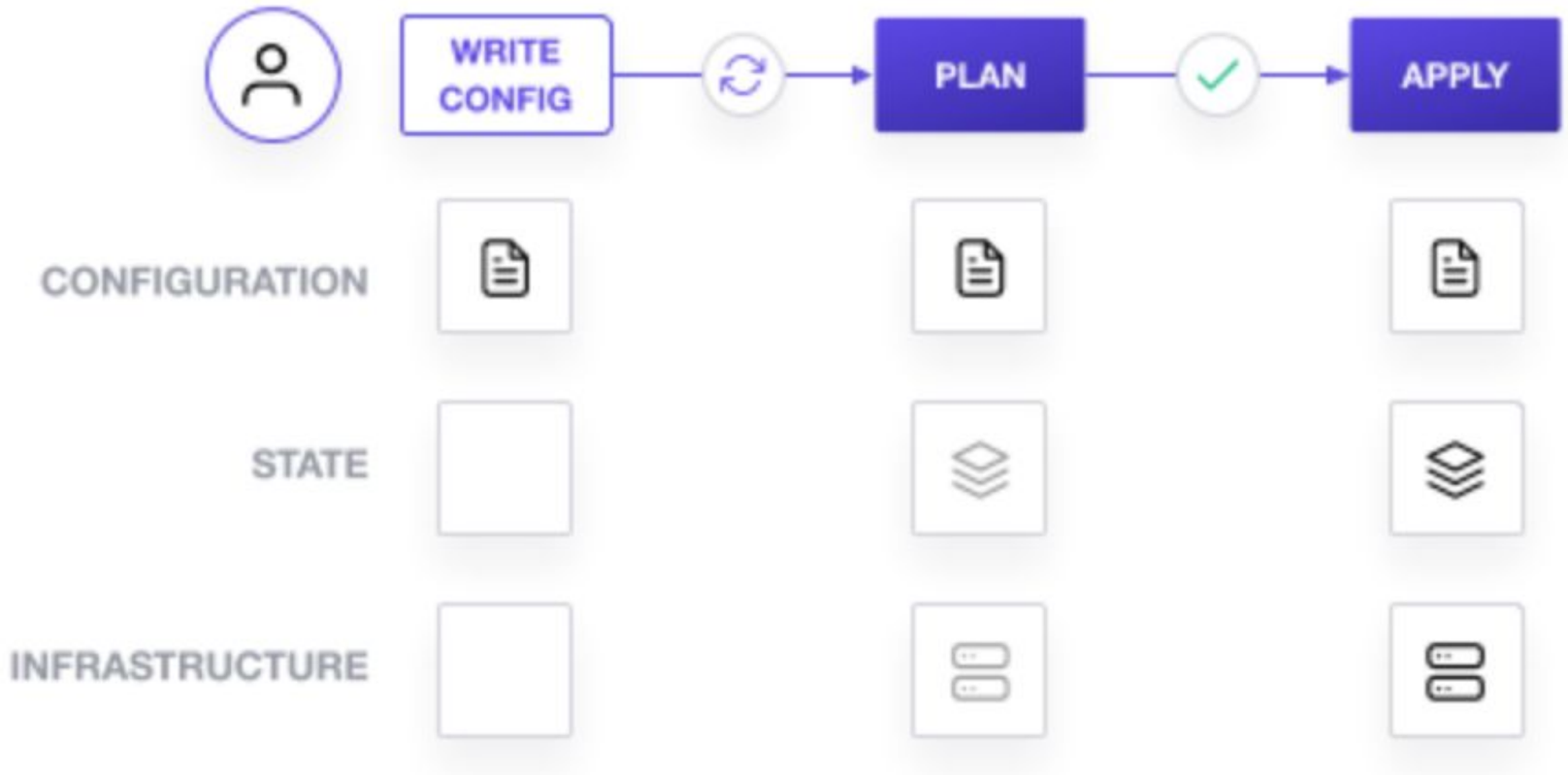
Init: Initializes the working directory which consists of all the configuration files.

Plan: Create an execution plan to reach a desired state of the infrastructure. Changes in the configuration files are done in order to achieve the desired state.

Apply: Makes the changes in the infrastructure as defined in the plan, and the infrastructure comes to the desired state.

Destroy: Used to delete all infrastructure resources marked after the apply phase.

WORKFLOW


WRITE CONFIG
PLAN
APPLY
CONFIGURATION
STATE
INFRASTRUCTURE

CORE CONCEPTS

- **Variables**: Used as input-variables, it is key-value pair used by modules to allow customization.
- **Provider**: It is a plugin to interact with APIs of service and access its related resources.
- **Module**: It is a folder with Terraform templates where all the configurations are defined.
- **State**: Cached information about the infrastructure managed by TF and the related configurations.
- **Data Source**: It is implemented by providers to return information on external objects to terraform.
- **Resources**: It refers to a block of one or more infrastructure objects (compute instances, virtual networks, etc.), which are used in configuring and managing the infrastructure.

VARIABLES

Terraform uses its own configuration language, designed to allow concise descriptions of infrastructure, **this language is declarative, describing an intended goal** rather than the steps to reach that goal. The main purpose is declaring resources, all other features exist only to make the definition of resources more flexible and convenient.

- **Input** variables are like function arguments.
- **Output** values are like function return values.
- **Local** values are like a function's temporary local variables.

https://www.terraform.io/docs/configuration/variables.html

VARIABLES - INPUT

```
variable "image_id" {
  type = string
}
```

```
variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}
```

```
variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

VARIABLES - INPUT

Type constraints are created from a mixture of type keywords and type constructors.

The supported type keywords are:

`string` `number` `bool`

The type constructors allow you to specify complex types such as collections:

`list(<TYPE>)` `set(<TYPE>)` `map(<TYPE>)` `object({<ATTR NAME> = <TYPE>, ... })`

VARIABLES - OUTPUT

Output values are like the return values of a Terraform module. Resource instances managed by Terraform each export attributes whose values can be used elsewhere in configuration.

Output values are a way to expose some of that information to the user of your module.

```
output "instance_ip_addr" {
   value       = aws_instance.server.private_ip
   description = "The private IP address of the main server instance."
}
```

VARIABLES - LOCALS

Local values can be helpful to avoid repeating the same values or expressions multiple times in a configuration, but if overused they can also make a configuration hard to read.

```
locals {
  # Ids for multiple sets of EC2 instances, merged together
  instance_ids = concat(aws_instance.blue.*.id, aws_instance.green.*.id)

  # Common tags to be assigned to all resources
  common_tags = {
    Service = local.service_name
    Owner   = local.owner
  }
}
```

RESOURCE

Resource declarations can include a number of advanced features, but only a small subset are required for initial use. More advanced syntax features, such as single resource declarations that produce multiple similar remote objects

```
resource "aws_instance" "web" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"
}
```

A resource block declares a resource of a given type ("aws_instance") with a given local name ("web").

The name is used to refer to this resource from elsewhere in the same module, but has no significance outside that module's scope.

Type and name serve as an identifier for a given resource and so must be unique within a module.

PROVIDERS

Terraform relies on **plugins called "providers"** to interact with remote systems. Configurations must declare which providers they require, so that Terraform can install and use them. Additionally, some providers require configuration (like endpoint URLs or cloud regions) before they can be used.

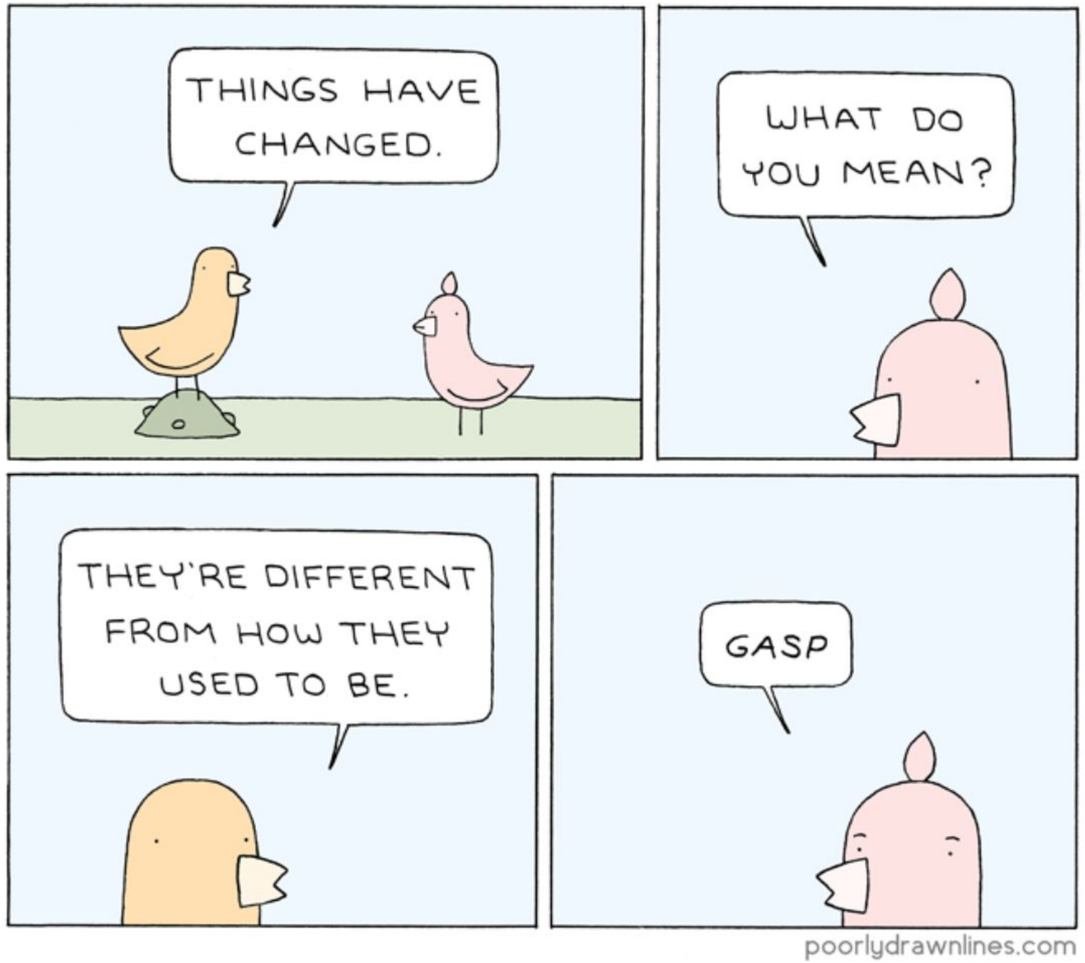
```
provider "google" {
  project = "acme-app"
  region  = "us-central1"
}
```

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

# Additional provider configuration
provider "aws" {
  alias  = "west"
  region = "us-west-2"
 }
```

DATA SOURCE

Data sources allow data to be fetched or computed for use elsewhere in Terraform configuration. Use of data sources allows a Terraform configuration to make use of information defined outside of Terraform, or defined by another separate Terraform configuration.

```
data "aws_ami" "example" {
  most_recent = true

  owners = ["self"]
  tags = {
    Name   = "app-server"
    Tested = "true"
  }
}
```

Thanks!

Catherin Cruz
DevOps Community
DevWeek Nov/2020