TEKsystems Global Services | WORKFORCE DEVELOPMENT

Configuration Management and DevOps: Automate Infrastructure Deployments

PARTICIPANT GUIDE

Content Usage Parameters

Content refers to material including instructor guides, student guides, lab guides, lab or hands-on activities, computer programs, etc. designed for use in a training program

1

Content is subject to copyright protection

2

Content may only be leveraged by students enrolled in the training program

3

Students agree not to reproduce, make derivative works of, distribute, publicly perform and publicly display content in any form or medium outside of the training program

4

Content is intended as reference material only to supplement the instructor-led training

Terraform Developer

Hi!

Jason Smith

Cloud Consultant with a Linux sysadmin background.
Focused on cloud-native technologies: automation, containers & orchestration

LinkedIn
https://www.linkedin.com/in/jruels/

mail
jason@innovationinsoftware.com

github
https://github.com/jruels

Expertise

- Cloud
- Automation
- CICD
- Docker
- Kubernetes

Introductions

- Name
- Job Role
- Your experience with
  - Infrastructure as Code
  - Ansible
  - Terraform
- Expectations for course (please be specific)

https://jruels.github.io/tf-dev

Configuration Management

Configuration Drift

POTHOLE

- Your infrastructure requirements change
- Configuration of a server falls out of policy

- Manage with Infrastructure as Code (IAC)
  - Terraform
  - Ansible
  - Chef
  - Puppet

Manual

POP QUIZ: Manual tasks

What manual tasks do you deal with repeatedly?

Can they be automated?

Common manual tasks

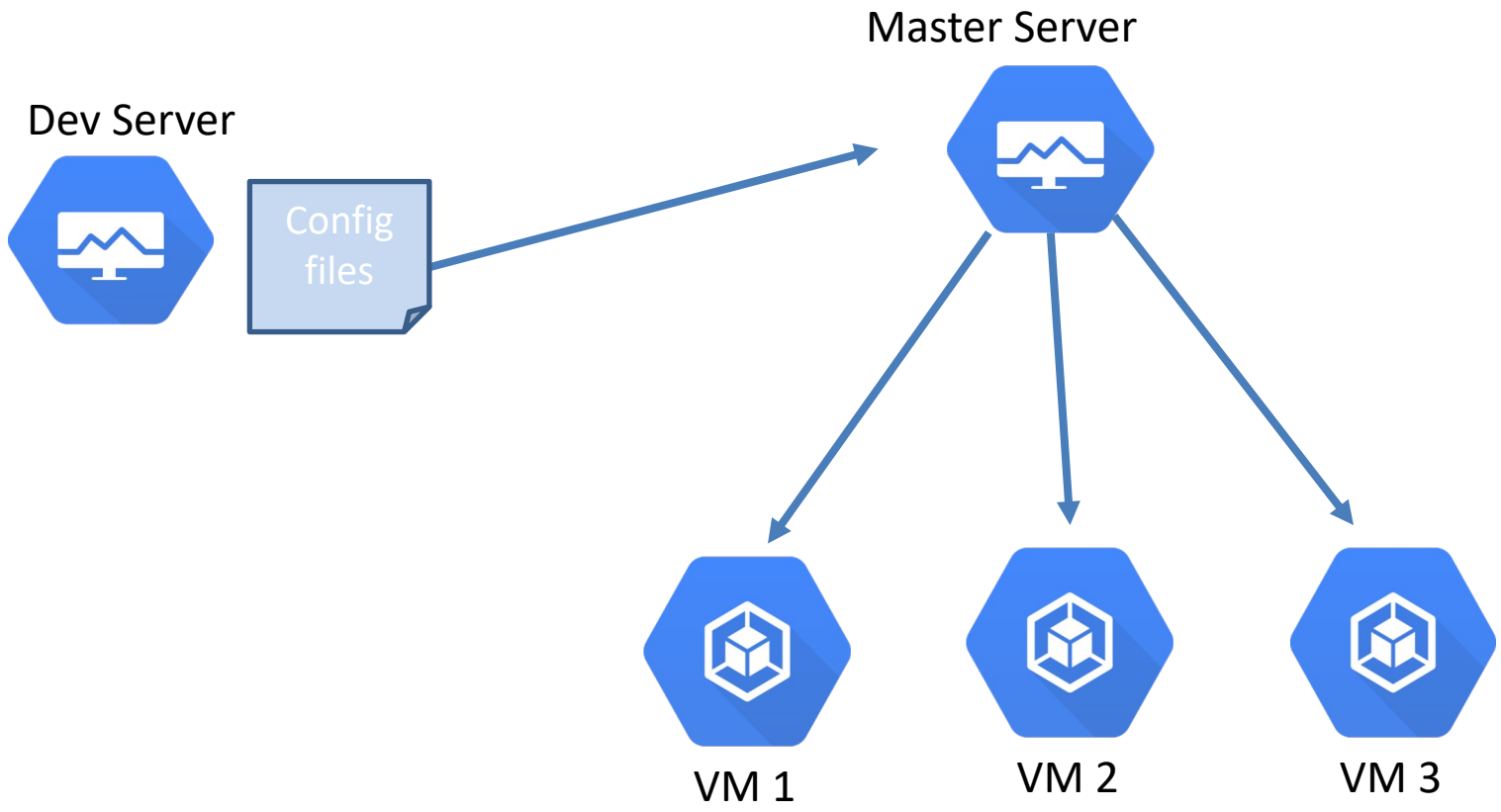
Installation and configuration:

- Operating System
- Runtime
- Libraries
- Utilities
- Configuration files

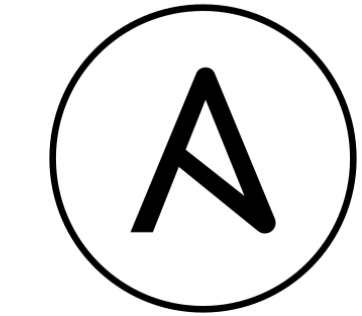
- Build application
- Test application
- Deploy application

Automated

Automated



Automation tools



Infrastructure as Code

What is IaC?

Infrastructure as code (IaC) is the process of managing and provisioning computer data centers through machine-readable definition files
Used with bare-metal as well as virtual machines and many other resources.
Normally a declarative approach

Infrastructure as Code

- Programmatically provision and configure components
- Treat like any other code base
  - Version control
  - Automated testing
  - data backup

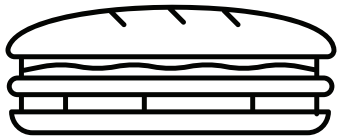
Infrastructure as Code

Provisioning infrastructure through software to achieve consistent and predictable environments.

Core Concepts

- Defined in code
- Stored in source control
- Declarative or imperative

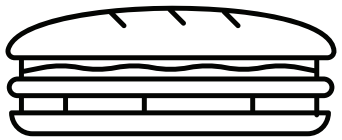
Imperative



```
# Make a sandwich
get bread
get mayo
get turkey
get lettuce

spread mayo on bread
put lettuce in between bread
put turkey in between bread
on top of turkey
```

Declarative



```
# Make a sandwich
food sandwich "turkey" {
  ingredients = [
    "bread", "turkey",
"mayo", "lettuce"
  ]
}
```

POP QUIZ:

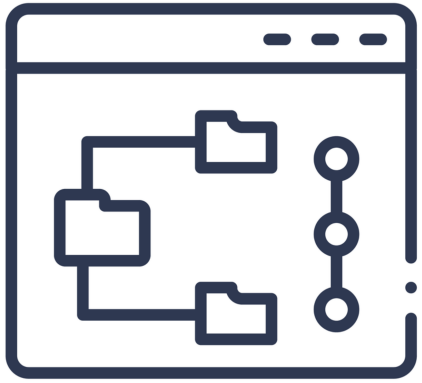
What challenges does Infrastructure as Code solve?

Demo: Automation Tools

Terraform Introduction

Automating Infrastructure

Provisioning resources

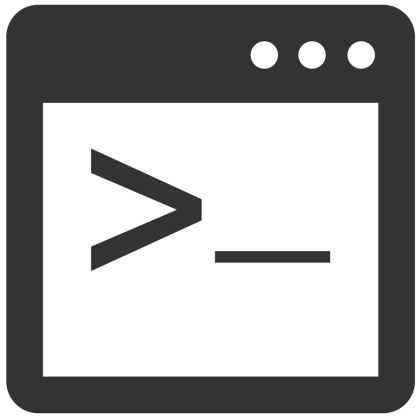


Version Control

Plan Updates

Reusable Templates

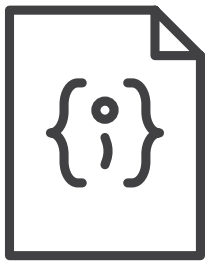
Terraform components



Terraform executable

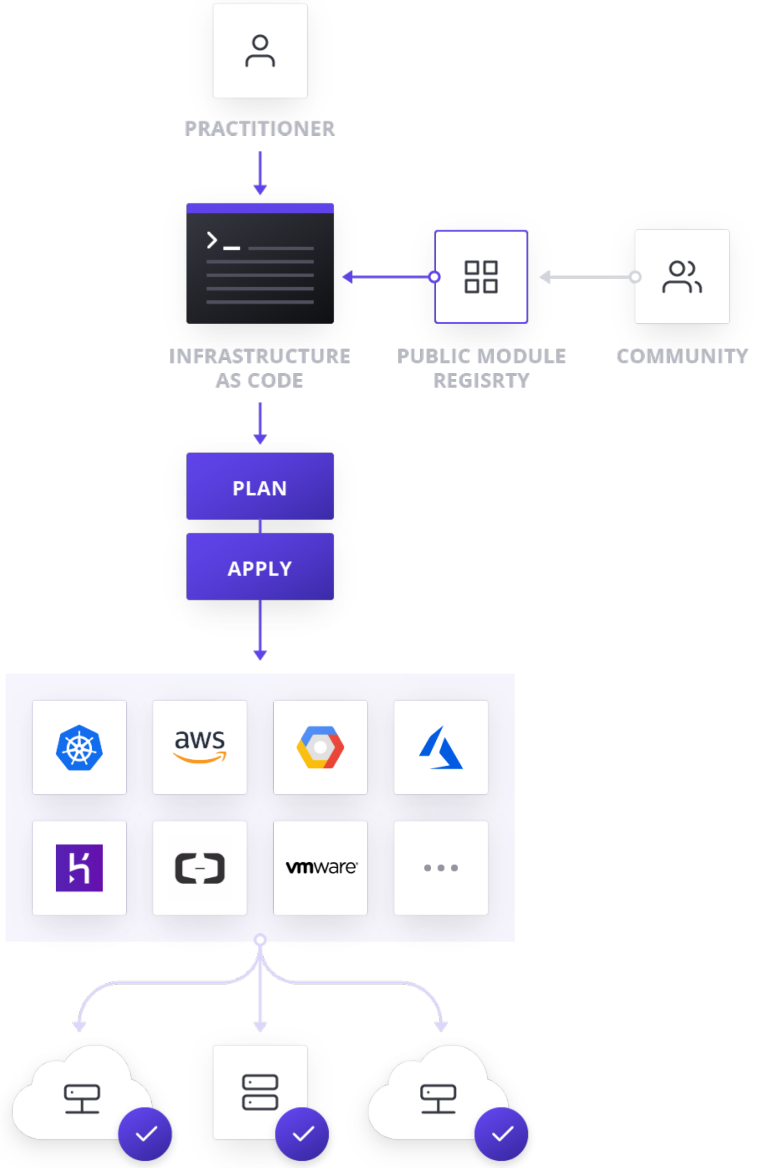
Terraform files

Terraform plugins



Terraform state

Terraform architecture



Terraform files

```
##Amazon Infrastructure
provider "aws" {
  region = "${var.aws_region}"
}
```

- Provider defines what infrastructure Terraform will be managing
  - Pass variables to provider
    - Region, Flavor, Credentials

Terraform files

```
data "aws_ami" "ubuntu" {
  most_recent = true
  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-*"]
  }
  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
  owners = ["099720109477"] # Canonical
}
```

- Data sources
  - Queries AWS API for latest Ubuntu 16.04 image.
  - Stores results in a variable which is used later.

Terraform files

```
resource "aws_instance" "aws-k8s-master" {
  subnet_id              = "${var.aws_subnet_id}"
  depends_on             = ["aws_security_group.k8s_sg"]
  ami                    = "${data.aws_ami.ubuntu.id}"
  instance_type          = "${var.aws_instance_size}"
  vpc_security_group_ids = ["${aws_security_group.k8s_sg.id}"]
  key_name               = "${var.aws_key_name}"
  count                  = "${var.aws_master_count}"
  root_block_device {
    volume_type           = "gp2"
    volume_size           = 20
    delete_on_termination = true
  }
  tags {
    Name = "k8s-master-${count.index}"
    role = "k8s-master"
  }
}
```

- Resource: Defines specifications for creation of infrastructure.

Terraform files

```
resource "aws_instance" "aws-k8s-master" {
  subnet_id              = "${var.aws_subnet_id}"
  depends_on             = ["aws_security_group.k8s_sg"]
  ami                    = "${data.aws_ami.ubuntu.id}"
  instance_type          = "${var.aws_instance_size}"
  vpc_security_group_ids = ["${aws_security_group.k8s_sg.id}"]
  key_name               = "${var.aws_key_name}"
  count                  = "${var.aws_master_count}"
  root_block_device {
    volume_type           = "gp2"
    volume_size           = 20
    delete_on_termination = true
  }
  tags {
    Name = "k8s-master-${count.index}"
    role = "k8s-master"
  }
}
```

- ami is set by results of previous data source query

Terraform files

```
##AWS Specific Vars
variable "aws_master_count" {
  default = 10
}
variable "aws_worker_count" {
  default = 20
}
variable "aws_key_name" {
  default = "k8s"
}
variable "aws_instance_size" {
  default = "t2.small"
}
variable "aws_region" {
  default = "us-west-1"
}
```

- Define sane defaults in variables.tf

Terraform files

```
resource "aws_instance" "aws-k8s-master" {
  subnet_id              = "${var.aws_subnet_id}"
  depends_on             = ["aws_security_group.k8s_sg"]
  ami                    = "${data.aws_ami.ubuntu.id}"
  instance_type          = "${var.aws_instance_size}"
  vpc_security_group_ids = ["${aws_security_group.k8s_sg.id}"]
  key_name               = "${var.aws_key_name}"
  count                  = "${var.aws_master_count}"
  root_block_device {
    volume_type           = "gp2"
    volume_size           = 20
    delete_on_termination = true
  }
  tags {
    Name = "k8s-master-${count.index}"
    role = "k8s-master"
  }
}
```

- Value is 'count' is setup by variable in variables.tf

POP QUIZ:

What is Terraform used for?

Terraform CLI

Run `terraform` command

Output:

```
Usage: terraform [-version] [-help] <command> [args]

The available commands for execution are listed below.
The most common, useful commands are shown first, followed by
less common or more advanced commands. If you're just getting
started with Terraform, stick with the common commands. For the
other commands, please read the help and docs before usage.

Common commands:
    apply              Builds or changes infrastructure
    console            Interactive console for Terraform
interpolations
    destroy            Destroy Terraform-managed infrastructure
    env                Workspace management
    fmt                Rewrites config files to canonical format
    get                Download and install modules for the
configuration
    graph              Create a visual graph of Terraform
resources
```

Terraform CLI

Command:

```
terraform init
```

Output:

```
Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "docker"...
```

Terraform fetches any required providers and modules and stores them in the .terraform directory. Check it out and you'll see a plugins folder.

Terraform CLI

Command:

```
terraform validate
```

Validate all of the Terraform files in the current directory. Validation includes basic syntax check as well as all variables declared in the configuration are specified.

Terraform CLI

Command:

```
terraform plan
```

Output:

```
Terraform will perform the following actions:

  + aws_instance.aws-k8s-master
      id:                                   <computed>
      ami:                                  "ami-01b45..."
      instance_type:                        "t3.small"
```

Plan is used to show what Terraform will do if applied. It is a dry run and does not make any changes.

Terraform CLI

Command:

```
terraform apply
```

Output:

```
❯ terraform apply "rapid-app.out"
aws_security_group.k8s_sg: Creating...
  arn:                                      "" => "<computed>"
  description:                              "" => "Allow all
inbound traffic necessary for k8s"
  egress.#:                                 "" => "1"
  egress.482069346.cidr_blocks.#:           "" => "1"
  egress.482069346.cidr_blocks.0:           "" => "0.0.0.0/0"
```

Performs the actions defined in the plan

Terraform CLI

Command:

```
terraform state show <resource>
```

```
terraform state show aws_instance.lab2-tf-example
```

Output:

```
ami                                  = "ami-830c94e3"
availability_zone                    = "us-west-2b"
cpu_core_count                       = 1
...
```

Shows information about provided resource.

Terraform CLI

The `terraform refresh` command is used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure. This can be used to detect any drift from the last-known state, and to update the state file.
This does not modify infrastructure but does modify the state file. If the state is changed, this may cause changes to occur during the next `plan` or `apply`.

Terraform CLI

Command:

```
terraform destroy [-auto-approve]
```

Destroys all the resources in the state file.
-auto-approve (don't prompt for confirmation)

Lab: Build first instance

Terraform CLI

Terraform is normally run from inside the directory containing the *.tf files for the root module. Terraform checks that directory and automatically executes them.

In some cases, it makes sense to run the Terraform commands from a different directory. This is true when wrapping Terraform with automation. To support that Terraform can use the global option `-chdir=...` which can be included before the name of the subcommand.

Terraform CLI

The `chdir` option instructs Terraform to change its working directory to the given directory before running the given subcommand. This means that any files that Terraform would normally read or write in the current working directory will be read or written in the given directory instead.

Terraform CLI

Command:

```
terraform -chdir=environments/dev (apply|plan|destroy)
```

Output:

```
❯ terraform apply
aws_security_group.k8s_sg: Creating...
  arn:                                      "" => "<computed>"
  description:                              "" => "Allow all
inbound traffic necessary for k8s"
  egress.#:                                 "" => "1"
  egress.482069346.cidr_blocks.#:           "" => "1"
  egress.482069346.cidr_blocks.0:           "" => "0.0.0.0/0"
```

Performs the subcommand in the specified directory.

Terraform CLI

It can be time consuming to update a configuration file and run `terraform apply` repeatedly to troubleshoot expressions not working.

Terraform has a console subcommand that provides an interactive console for evaluating these expressions.

Terraform CLI

```
variable "x" {

default = [
        {
          name = "first",
          condition = {
            age = "1"
          }
          action = {
            type = "Delete"
          }
        },{
          name = "second",
          condition = {
            age = "2"
          }
          action = {
            type = "Delete"
          }
        }
    ]
}
```

Terraform CLI

Command:

```
terraform console
```

Output:

```
> var.x[1].name
"second"

var.x[0].condition
{
  "age" = "1"
}
```

Test expressions and interpolations interactively.

Terraform CLI

Terraform includes a graph command for generating visual representation of the configuration or execution plan. The output is in DOT format, which can be used by GraphViz to generate charts.

Terraform CLI

Command:

```
terraform graph
```

Output:

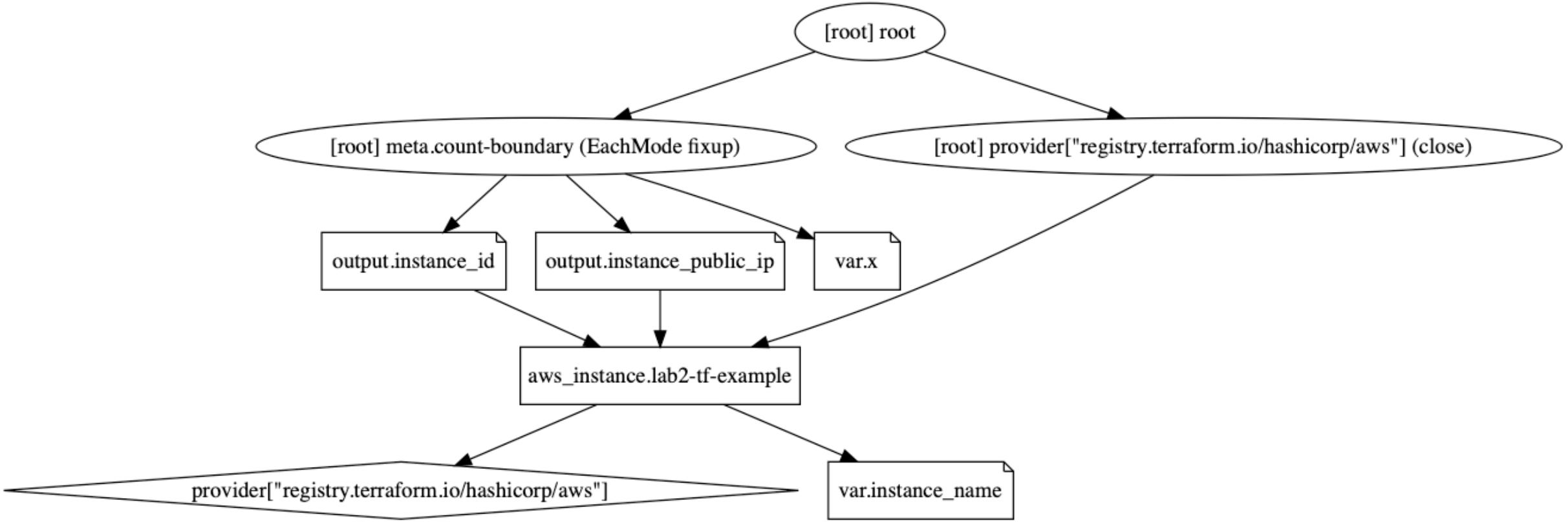
```
digraph {
        compound = "true"
        newrank = "true"
        subgraph "root" {
                "[root] aws_instance.lab2-tf-example (expand)"
[label = "aws_instance.lab2-tf-example", shape = "box"]
...
```

Create a visual graph of Terraform resources

```
terraform graph | dot -Tsvg > graph.svg
```

Terraform CLI

```
terraform graph| dot -Tsvg > graph.svg
```



Terraform vs JSON

ARM JSON:

```
"name" : "[concat(parameters('PilotServerName'), '3')]",
```

Terraform:

```
name = "${var.PilotServerName}3"
```

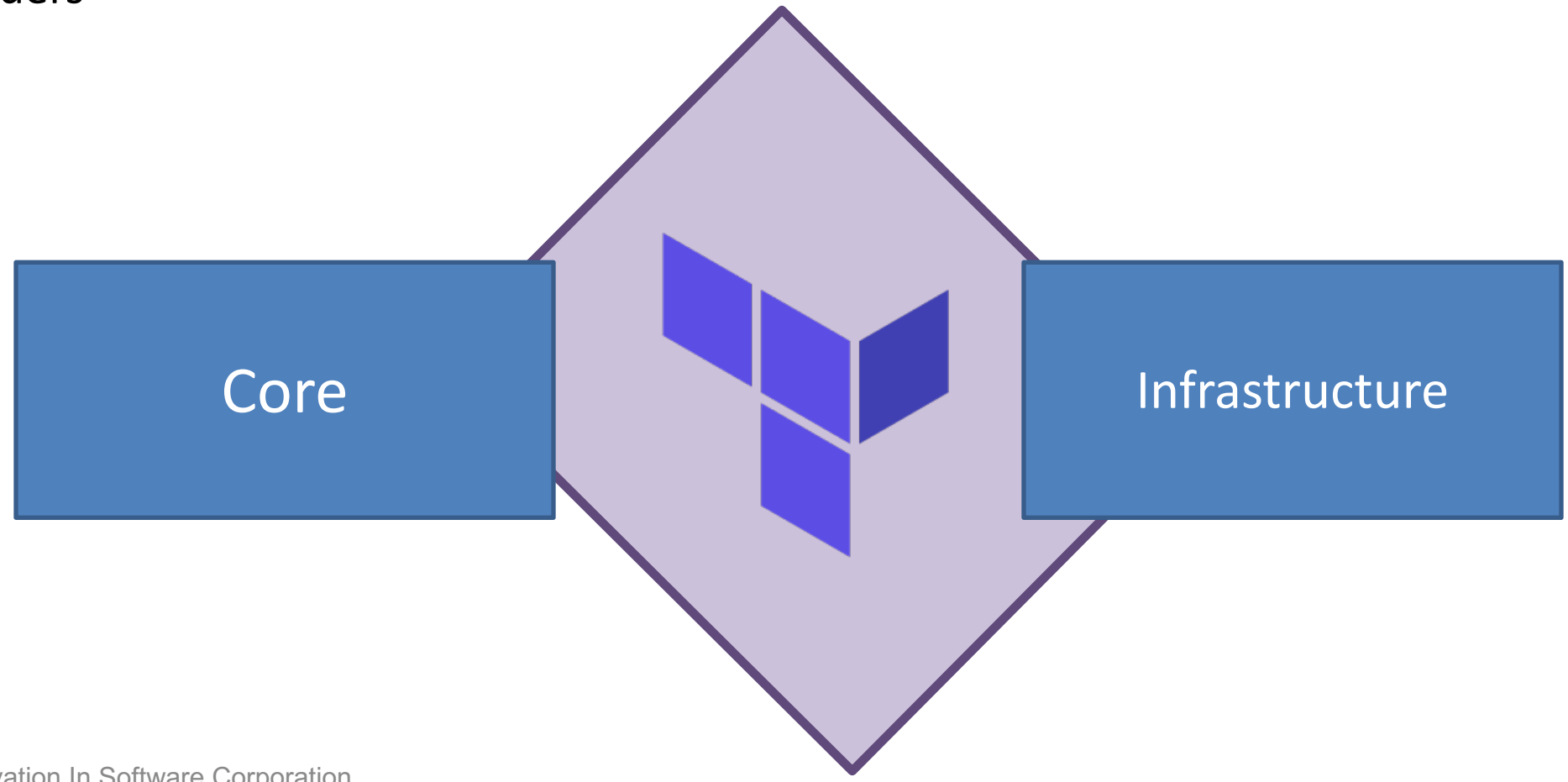
Terraform code (HCL) is "easy" to learn and "easy" to read. It is also more compact than equivalent JSON configuration.

terraform model

Terraform code is broken into three parts: Core Terraform, Modules and Providers

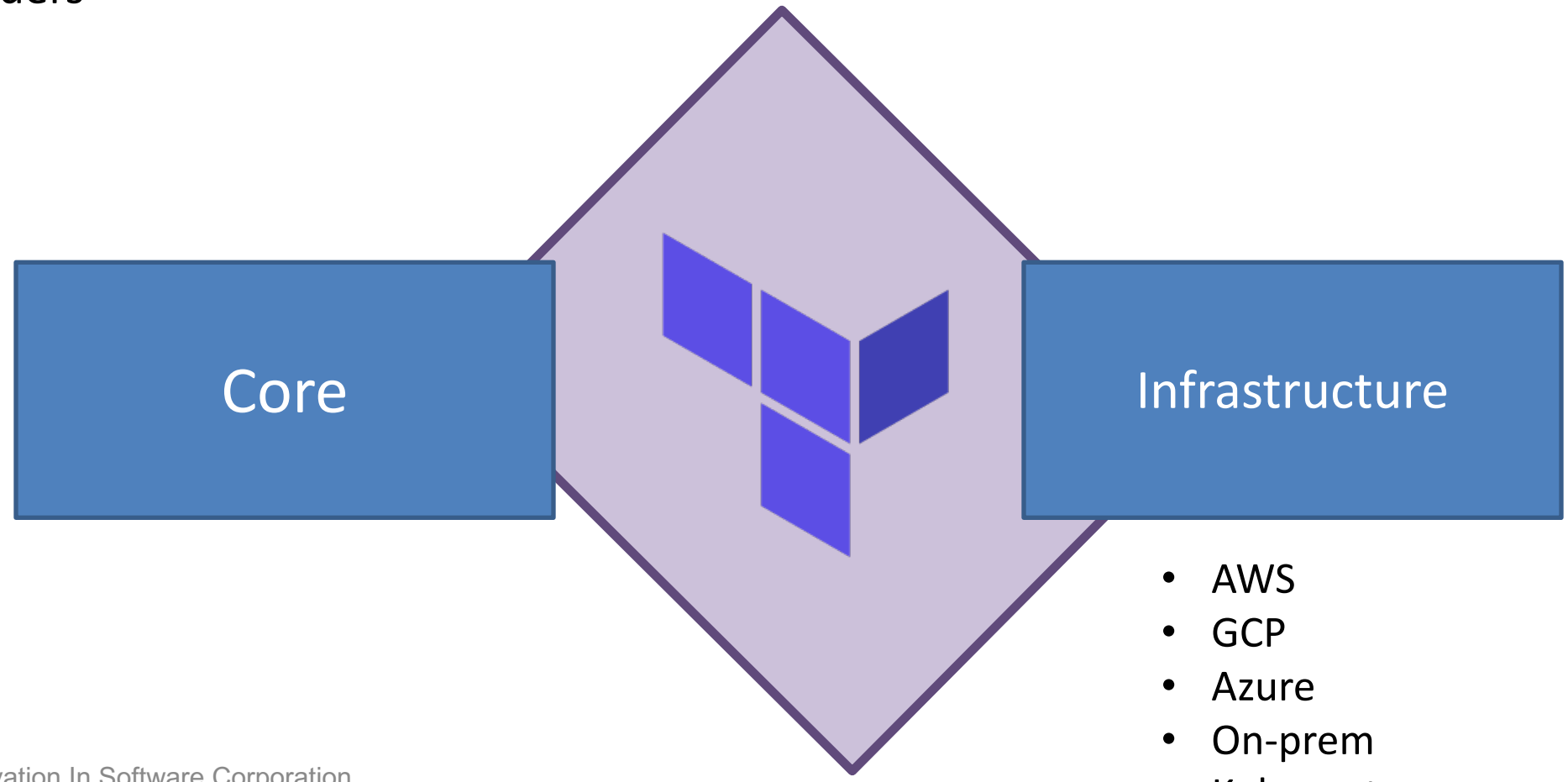
terraform model

Terraform code is broken into three parts: Core Terraform, Modules and Providers



terraform model

Terraform code is broken into three parts: Core Terraform, Modules and Providers



- AWS
- GCP
- Azure
- On-prem
- Kubernetes

configuration files

Terraform has a pretty confusing file hierarchy. We are going to start with the basics.

`provider.tf` - specify provider information

`main.tf` - Create infrastructure resources.

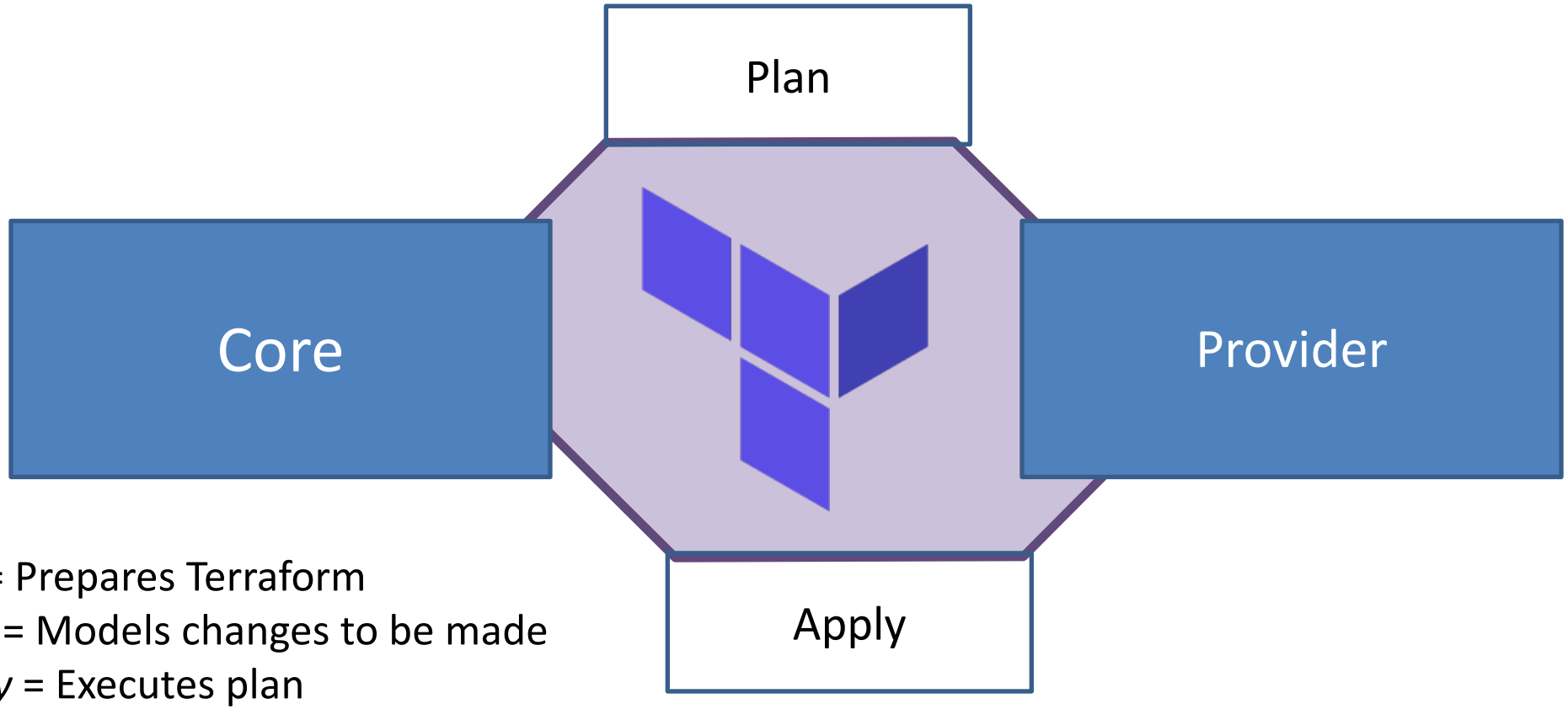
`variables.tf` - define values for variables in main.tf

Terraform files

- core = Terraform language, logic, and tooling.
- provider = Pluggable code to allow Terraform to talk to vendor APIs
- modules = A container for multiple resources that are used together.

terraform workflow

Terraform workflow consists of three common commands, Init, Plan and Apply.

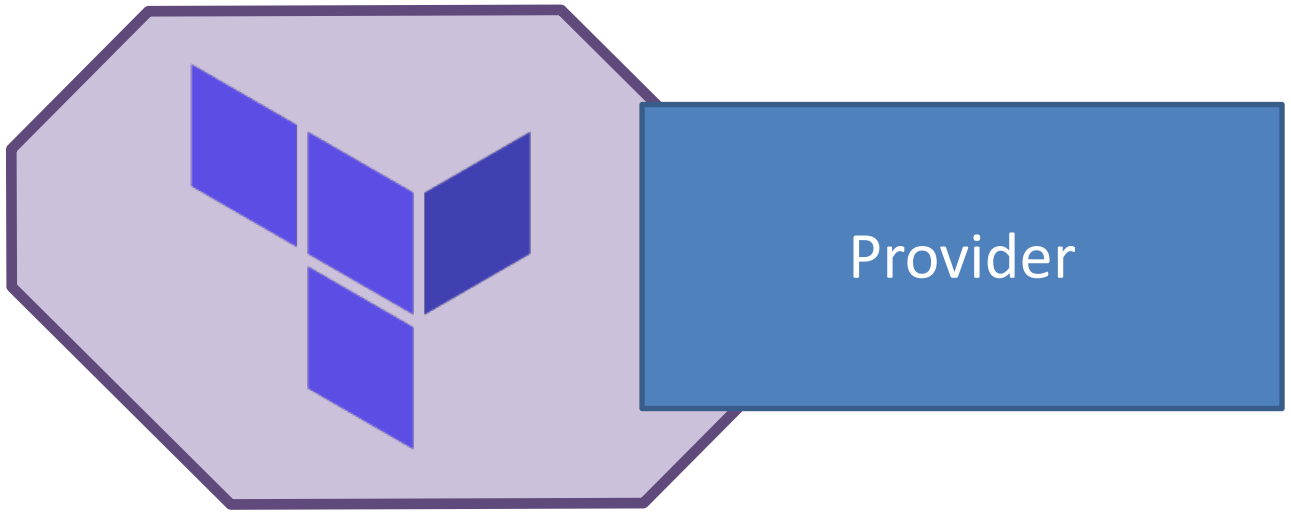


- *init* = Prepares Terraform
- *plan* = Models changes to be made
- *apply* = Executes plan

terraform providers

Terraform has over 150 "supported" providers.

- Clouds (AWS, GCP, Rackspace, Azure etc.)
- Version control (GitHub, GitLab, Bitbucket)
- Software (Grafana, Consul, Docker, Kubernetes)
- more!



Terraform providers

Terraform relies on plugins called "providers" to interface with remote systems.

Terraform 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.

Terraform providers

Each provider adds resources and/or data sources that Terraform manages.

Every resource type is implemented by a provider. Terraform can't manage any kind of infrastructure without providers.

Providers are used to configure infrastructure platforms (either cloud or self-hosted). Providers also offer utilities for tasks like generating random numbers for unique resource names.

Terraform files

```
##Amazon Infrastructure
provider "aws" {
  region = "${var.aws_region}"
}
```

- Provider defines what infrastructure Terraform will be managing
  - Pass variables to provider
    - Region, Flavor, Credentials

Terraform provider configuration

```
# The default provider configuration; resources that begin with `aws_` will use
# it as the default, and it can be referenced as `aws`.
provider "aws" {
  region = "us-east-1"
}

# Additional provider configuration for west coast region; resources can
# reference this as `aws.west`.
provider "aws" {
  alias  = "west"
  region = "us-west-2"
}
```

Terraform supports 'aliases' for multiple configurations for the same provider, and you can select which one to use on a per-resource or per-module basis.

Terraform provider configuration

```
provider "azurerm" {
  version = "=1.30.1"
}
```

Terraform allows you to configure the providers in code. configure specific versions or pull in latest.

Terraform provider configuration

```
provider "azurerm" {
  version = "=1.30.1"
  subscription_id = "SUBSCRIPTION-ID"
  client_id       = "CLIENT-ID"
  client_secret   = "CLIENT-SECRET"
  tenant_id       = "TENANT_ID"
}
```

Providers allow you to authenticate in the code block. **This is a very BAD idea.**

Terraform provider configuration

For Azure that can be az login, Managed Service Identity, or environment variables.

```
az login
```

```
export ARM_TENANT_ID=
export ARM_SUBSCRIPTION_ID=
export ARM_CLIENT_ID=
export ARM_CLIENT_SECRET=
```

Terraform provider configuration

Terraform will use the native tool's method of authentication. Environment variables, shared credentials files or static credentials.

```
provider "aws" {}
```

```
export AWS_ACCESS_KEY_ID=
export AWS_SECRET_ACCESS_KEY=
```

Terraform provider configuration

Terraform will use the native tool's method of authentication. Environment variables, shared credentials files or static credentials.

```
provider "aws" {
  region                  = "us-west-2"
  shared_credentials_file = "~/.aws/creds"
  profile                 = "customprofile"
}
```

Terraform provider configuration

Terraform will use the native tool's method of authentication. Environment variables, shared credentials files or static credentials.

```
provider "aws" {
  access_key     = "MYKEY"
  secret_key     = "MYSECRET"
}
```

Terraform ASA provider

```
provider "ciscoasa" {
  api_url           = "https://10.0.0.5"
  username          = "admin"
  password.         = "YOUR SECRET PASSWORD"
  ssl_no_verify     = false
}
```

Providers allow you to authenticate in the code block. **This is a very BAD idea.**

Use environment variables:

`CISCOASA_USER`
`CISCOASA_PASSWORD`

Terraform ASA ACL resource

```
resource "ciscoasa_acl" "foo" {
  name = "aclname"
  rule {
    source                = "192.168.10.5/32"
    destination           = "192.168.15.0/25"
    destination_service = "tcp/443"
  }
  rule {
    source                = "192.168.10.0/24"
```

Example of creating ACL

Terraform ASA static route resource

```
resource "ciscoasa_static_route" "ipv4_static_route" {
  interface               = "inside"
  network                 = "10.254.0.0/16"
  gateway                 = "192.168.10.20"
  }

resource "ciscoasa_static_route" "ipv6_static_route" {
  interface               = "inside"
  network                 = "fd01:1337::/64"
  gateway                 = "fd01:1338::1"
```

Terraform resources

Resources are the most important element of the Terraform language. Each resource block describes one or more infrastructure objects, such as networks, instances, or higher-level objects such as DNS records.

Terraform resources

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 Terraform module but has no significance outside that module's scope.

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

Terraform resources

Every Terraform resource is structured the exact same way.

```
resource "type" "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
```

resource = top level keyword

Terraform resources

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

Configuration data for the "aws_instance" resource:

- AMI
- Instance Type (flavor)

Terraform resources

Within the block body (between { and }) are the configuration arguments for the resource itself. Most arguments in this section depend on the resource type, and indeed in this example both ami and instance_type are arguments defined specifically for the aws_instance resource type.

Terraform resources

Every Terraform resource is structured the exact same way.

```
resource "type" "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
```

type = this is the name of the resource. The first part tells you which provider is belongs to. Example: `azurerm_virtual_machine`. This means the provider is Azure and the specific type of resources is a virtual machine.

Terraform resources

Every Terraform resource is structured the exact same way.

```
resource "type" "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
```

name = arbitrary name to refer to resource. Used internally by TF and cannot be a variable.

resources (building blocks)

Create an Azure Resource Group.

- Azure requires all resources be assigned to a resource group.

```
resource "azurerm_resource_group" "training" {
  name     = "training-workshop"
  location = "westus"
}
```

resources (building blocks)

TIP: You can assign random names to resources.

```
resource "random_id" "project"name" {
  byte_length = 4
}
resource "azurerm_resource_group" "training" {
  name       = "${random_id.project_name.hex}-training"
  location   = "westus"
}
```

Terraform resources

Some resource types provide a special timeouts nested block argument that allows you to customize how long certain operations are allowed to take before being considered to have failed. For example, aws_db_instance allows configurable timeouts for create, update and delete operations.

Terraform resources

Timeouts are handled entirely by the resource type implementation in the provider, but resource types offering these features follow the convention of defining a child block called timeouts that has a nested argument named after each operation that has a configurable timeout value. Each of these arguments takes a string representation of a duration, such as "60m" for 60 minutes, "10s" for ten seconds, or "2h" for two hours.

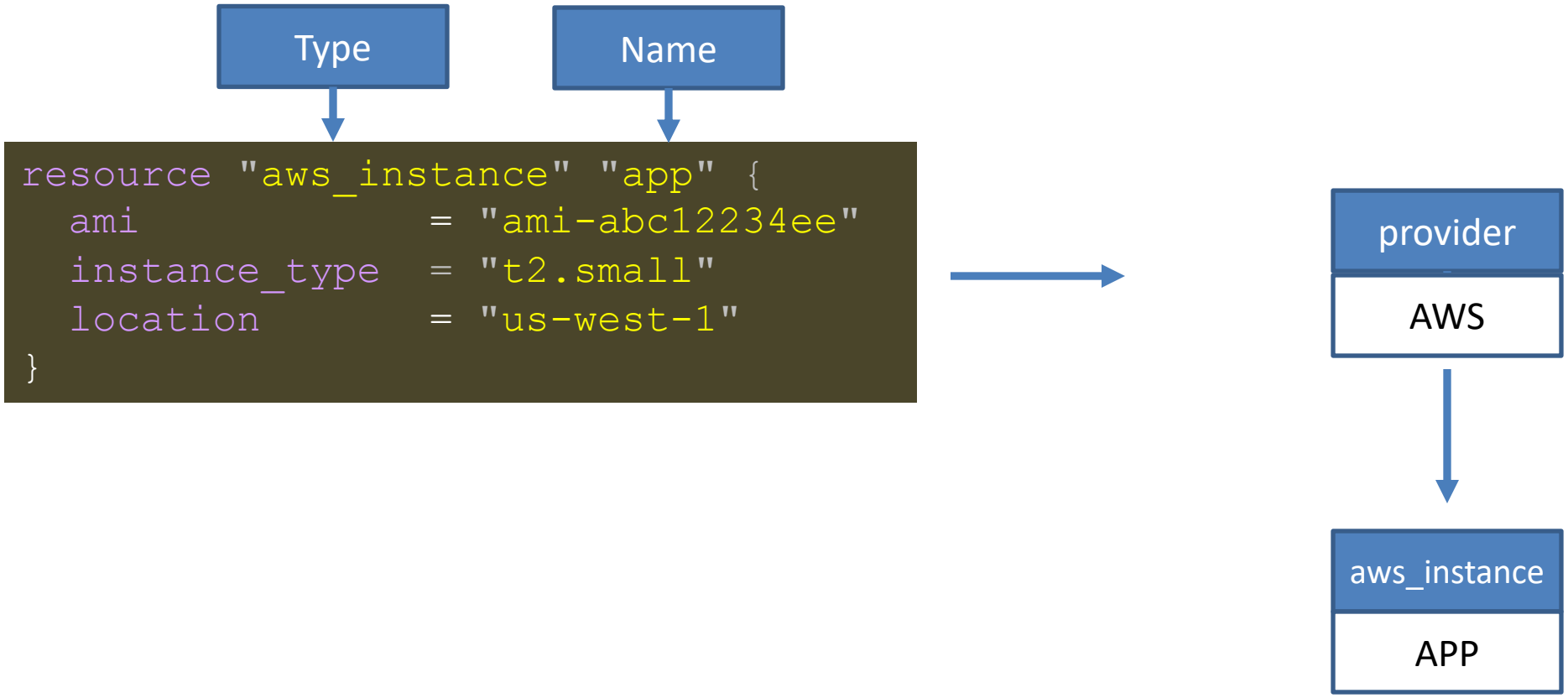
Terraform resources

```
resource "aws_db_instance" "example" {
  # ...

  timeouts {
    create = "60m"
    delete = "2h"
  }
}
```

Timeout arguments

resources (building blocks)



Terraform local-only resources

While most resource types correspond to an infrastructure object type that is managed via a remote network API, there are certain specialized resource types that operate only within Terraform itself, calculating some results and saving those results in the state for future use.

For example, you can use local-only resources for:

- Generating private keys
- Issue self-signed TLS certs
- Generating random ids

- The behavior of local-only resources is the same as all other resources, but their result data exists only within the Terraform state. "Destroying" such a resource means only to remove it from the state, discarding its data.

Terraform local-only resources

Local-only resources are also referred to as logical resources

This is primarily used for easy bootstrapping of throwaway development environments.

```
resource "tls_private_key" "example" {
  algorithm   = "ECDSA"
  ecdsa_curve = "P384"
}
```

Terraform variables

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

When you declare variables in the root module of your configuration, you can set their values using CLI options and environment variables. When you declare them in child modules, the calling module should pass values in the module block.

Terraform variables

Each input variable accepted by a module must be declared using a variable block.

The label after the variable keyword is a name for the variable, which must be unique among all variables in the same module. This name is used to assign a value to the variable from outside and to reference the variable's value from within the module.

```
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"
    }
  ]
}
```

Terraform variables

Terraform has some reserved variables:

- source
- version
- providers
- count
- for_each
- lifecycle
- depends_on
- locals

- These are reserved for meta-arguments in module blocks.

```
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"
    }
  ]
}
```

Terraform variables

Optional variable arguments:

- default: A default value which makes the variable optional
- type: Specifies value type accepted for the variable.
- description: Specifies the variable's documentation
- validation: A block to define validation rules
- sensitive: Does not show value in output

Terraform variables

Set default values for variables in `variables.tf`
If default values are omitted, user will be prompted.

```
##AWS Specific Vars
variable "aws_master_count" {
  default = 10
}
variable "aws_worker_count" {
  default = 20
}
variable "aws_key_name" {
  default = "k8s"
}
```

Terraform variables (type constraint)

Type constraints are optional but highly encouraged. They can serve as reminders for users of the module and help Terraform return helpful errors if the wrong type is used.

The type argument allows you to restrict acceptable value types. If no type constraint is defined, then a value of any type is accepted.

The keyword any may be used to indicate that any type is acceptable.

If both the type and default arguments are specified, the given default value must be convertible to the specified type.

Terraform variables (type constraint)

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

Supported type keywords:
- string
- number
- bool

- Type constructors allow you to specify complex types such as collections:
- list (<TYPE>)
- set (<TYPE>
- map (<TYPE>)
- object ({<ATTR NAME> = <TYPE>, ...})
- tuple ([<TYPE>, ...])

Terraform variables

Input variables are part of its user interface. You can briefly describe the purpose of each variable using the optional description argument.

```
## Variable description example
variable "image_id" {
  type = string
  description = "The id of the machine image (AMI)"
}
```

Terraform variables (validation)

In addition to Type Constraints, you can specify arbitrary custom validation rules for a variable.

```
## Variable validation example
variable "image_id" {
  type = string
  description = "The id of the machine image (AMI)"
}

  validation {
    condition = length(var.image_id) > 4 && substr(var.image_id, 0, 4) == "ami-"
     error_message = "The image_id value must be a valid AMI id, starting with
\"ami-\"."
  }
}
```

Terraform variables (validation)

A condition argument is an expression that must use the value of the variable to return true if the value is valid, or false if it is invalid.

The expression can refer only to the variable that the condition applies to and must not produce errors.

Terraform variables

If the failure of an expression is the basis of the validation decision, use the can function to detect such errors. For example:

```
## Variable validation example
variable "image_id" {
  type = string
  description = "The id of the machine image (AMI)"
}

  validation {
    # regex(...) fails if it cannot find a match
    condition. = can(regex("^ami-", var.image_id))
    error_message = "The image_id value must be valid."
```

Terraform variables

The `variables.tf` file includes values for variables.

```
variable "region" {
  description = "Region to deploy Jenkins into"
  default     = "us-west2"
}

variable "jenkins_initial_password" {
  description = "Jenkins user password"
  default     = "bitnami"
}

variable "project_id" {
  description = "Project for Jenkins VM"
}
```

Terraform variables

Variables can be referenced in a module.

```
data "template_file" "jenkins_startup_script" {
  template = local.jenkins_startup_script_template

  vars = {
    jenkins_username                = local.jenkins_username
    jenkins_password                = local.jenkins_password
    jenkins_workers_project_id      = var.jenkins_workers_project_id
    jenkins_workers_instance_cap    = var.jenkins_workers_instance_cap
    jenkins_workers_description     = var.jenkins_workers_description
  ..snip
}
```

Terraform variables

Variables can be also referenced in the root `main.tf` config file.

```
# Install Jenkins
module "jenkins" {
  source  = "./modules/tf-gcp-jenkins"
  
# required variables
  jenkins_initial_password        = var.jenkins_initial_password
#  jenkins_initial_username       = var.jenkins_initial_username
  project_id          = var.project_id
  region                          = var.region
  jenkins_instance_network        = var.jenkins_instance_network
  jenkins_instance_subnetwork     = var.jenkins_instance_subnetwork
  jenkins_instance_zone           = var.jenkins_instance_zone
  jenkins_workers_network         = var.jenkins_workers_network
  jenkins_workers_project_id      = var.jenkins_workers_project_id
  jenkins_workers_region          = var.jenkins_workers_region
}
```

Terraform output values

Output values are like the return values of a Terraform module, and have several uses:

- A child module can use outputs to expose a subset of its resource attributes to a parent module.
- A root module can use outputs to print certain values in the CLI output after running terraform apply.

Terraform output values

Use outputs to expose information from child modules to parent modules.

Child module:

```
output "jenkins_instance_initial_username" {
  description = "The initial username assigned to the Jenkins instance's `user` username"
  value       = local.jenkins_username
}

output "jenkins_instance_initial_password" {
  description = "The initial password assigned to the Jenkins instance's `user` username"
  value       = local.jenkins_password
}
```

Root module (parent):

```
output "jenkins__initial_username" {
  description = "The initial username assigned to Jenkins"
  value       = module.jenkins.jenkins_instance_initial_username
}

output "jenkins_initial_password" {
  description = "The initial password assigned to Jenkins"
  value       = module.jenkins.jenkins_instance_initial_password
}
```

Terraform local values

A local value assigns a name to an expression, so you can use it multiple times within a module without repeating it.

```
locals {
  jenkins_password = coalesce(
    var.jenkins_initial_password,
    random_string.jenkins_password.result,
  )
  jenkins_startup_script_template = file("${path.module}/templates/jenkins_startup_script.sh.tpl")
  jenkins_username                = "user"
```

Set global variables with local variable values

```
vars = {
    jenkins_username = local.jenkins_username
    jenkins_password = local.jenkins_password
  }
}
```

Terraform local values

Local values can hide the actual values being used.
Use local values in moderation. They are helpful to avoid repeating the same values or expressions multiple times in a configuration, but if use too often can make a configuration hard to read by future maintainers.

Only use local values in situations where a single value or result is used in many places AND that value is likely to be changed in the future.

The ability to change the value in one location is the key advantage of local values.

Terraform overrides

Terraform configurations are meant to be easy to read. Using a variables.tf file allows everyone who is using the module to understand exactly what is going to happen.

What if you want to use automation to deploy you infrastructure into different environments though?

The solution is overrides. Your automation pipeline can generate override files on-the-fly prior to Terraform running.

Terraform overrides

Override files must include override in the name. Most common file names are:

- *_override.tf
- *_override.tf.json
- override.tf
- override.tf.json

Terraform overrides

If you have a Terraform configuration "example.tf" with the following:

```
resources "aws_instance" "web" {
  instance_type = "t2.micro"
  ami           = "ami-44534340"
}
```

...and you created a file "override.tf" containing:

```
resources "aws_instance" "web" {
  ami    = "foo"
}
```

Terraform overrides

Terraform will merge the latter into the former, behaving as if the original config had been:

```
## Variable validation example
resources "aws_instance" "web" {
  instance_type = "t2.micro"
  ami           = "foo"
}
```

Lab: Variables and output

Lab: Create multi-resource infrastructure

Terraform state file

Terraform is a *stateful* application.

- Keeps track of everything you build inside of a state file
  - terraform.tfstate
- State file is Terraform's source of truth

```
{
  "version": 3,
  "terraform_version": "0.12.29",
  "serial": 6,
  "lineage": "0a209e29-de63-9e87-2cd2-4f2071717cee",
  "modules": [
      {
         "path": [
            "root"
         ],
         "outputs": {
            "MySQL_Server_FQDN": {
               "value": "labtest1-mysql-server.azure.com
```

Terraform state file

Each Terraform configuration can specify a backend, which defines where and how operations are performed, where state snapshots are stored, etc.

Terraform state file

When starting out with Terraform the best approach is to stick with a local backend. It also makes sense to use a local backend if you are the only one managing the infrastructure.

Terraform state file

Backend configuration is only used by Terraform CLI.

Terraform Cloud and Terraform Enterprise always use their own state storage when performing Terraform runs, so they ignore any backend block in the configuration.

It's common to use Terraform CLI with Terraform Cloud, so best practice is to include a backend block in the configuration with a remote backend pointing to the relevant Terraform Cloud workspace(s)

Terraform state file

Terraform supports local and remote state file storage.

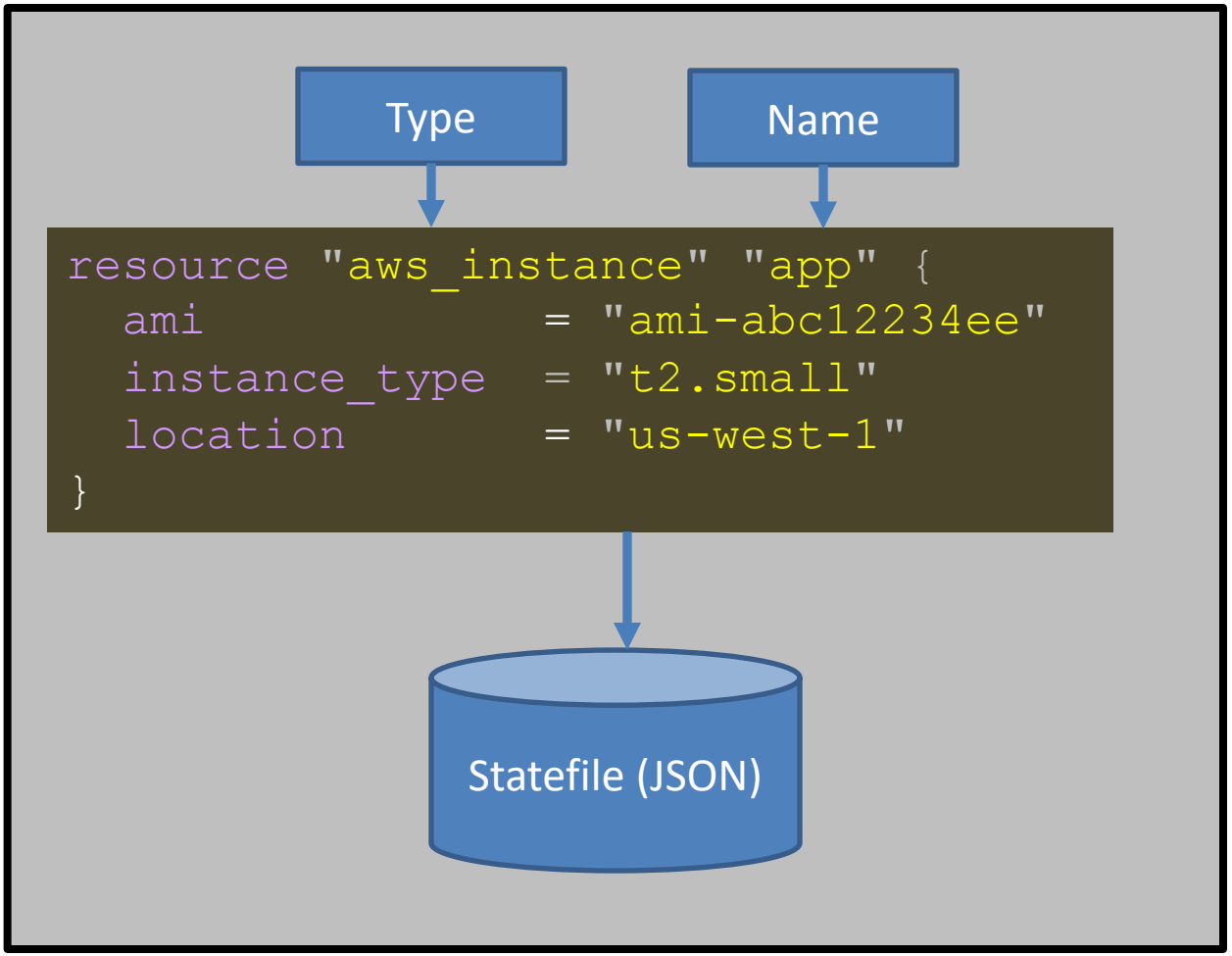
- Default is local storage
- Remote backends:
  - S3, Azure Storage, Google Cloud Storage

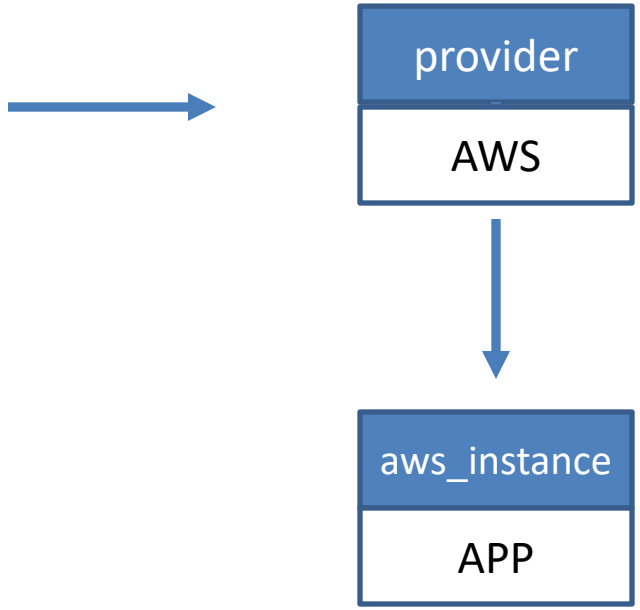
Terraform state file

Terraform supports many backends:

- local (default)
- remote (Terraform Enterprise/Terraform Cloud)
- azurerm
- consul
- s3
- gcs
- more...

Terraform state (local)



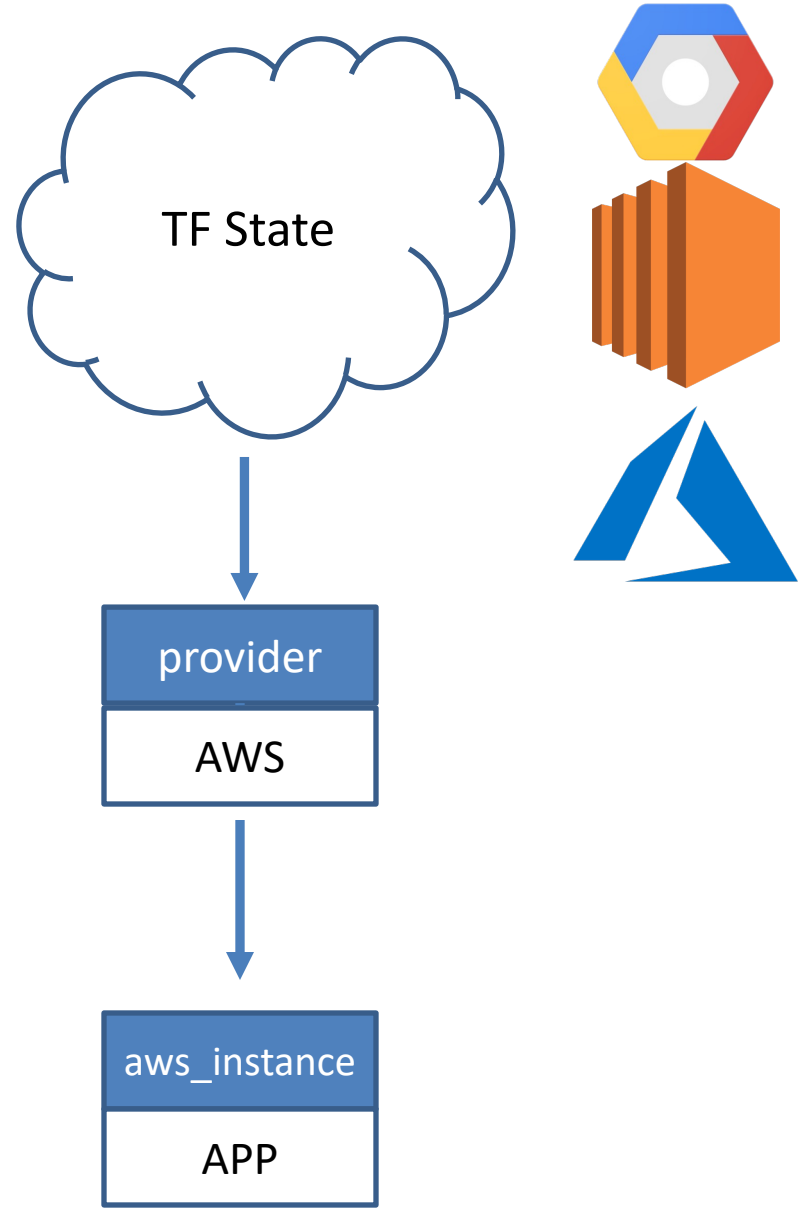


Terraform state (remote)

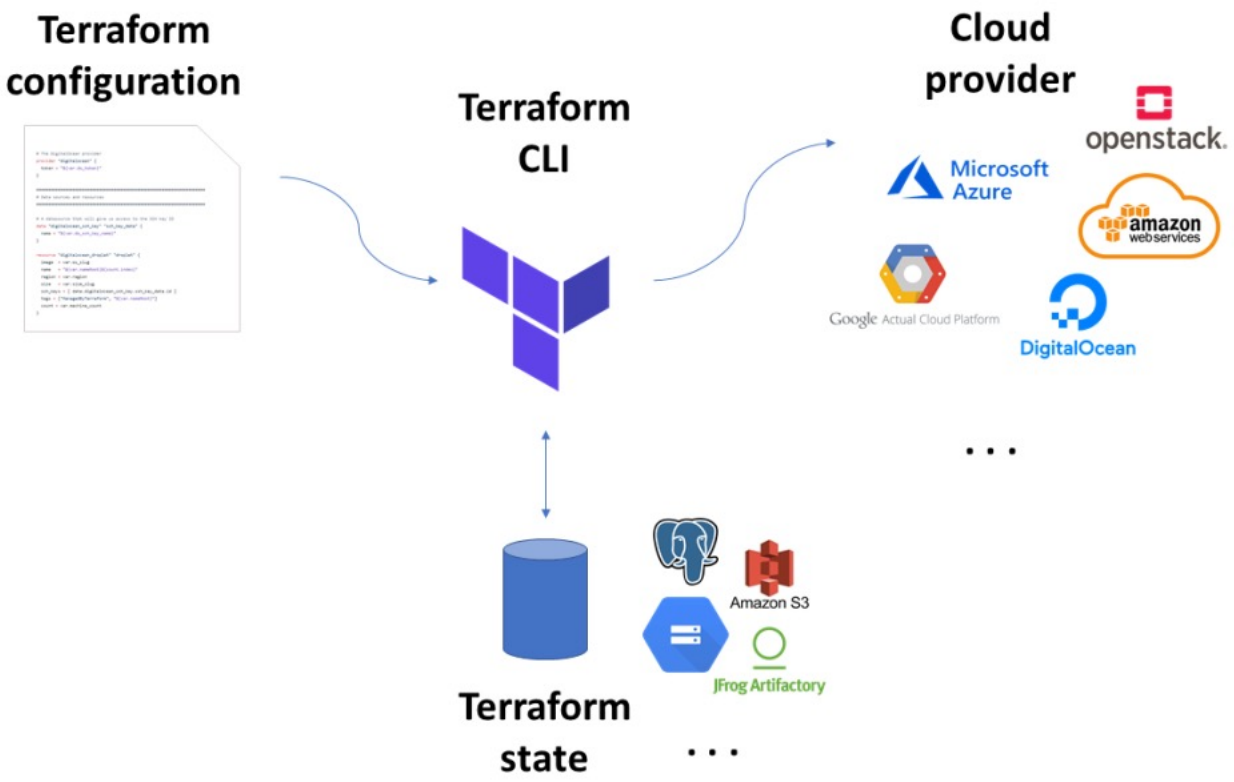
Type

Name

```
resource "aws_instance" "app" {
  ami             = "ami-abc12234ee"
  instance_type   = "t2.small"
  location        = "us-west-1"
}
```



Terraform state (remote)



Terraform state file locking

Terraform will lock your state for all operations that could write state if your backend supports it. This prevents others from acquiring the lock and possibly corrupting your state.

State locking automatically happens on all operations that write state.

- azurerm
- consul
- etcdv3
- gcs
- s3 (locking via DynamoDB)

Terraform state file

Backends are configured with a nested "backend" block within the top-level terraform block.

```
terraform {
  backend "remote" {
    organization              = "foo"
}
```

Lab: Implement bool, strings and numbers

Lab: Migrate to remote state backend