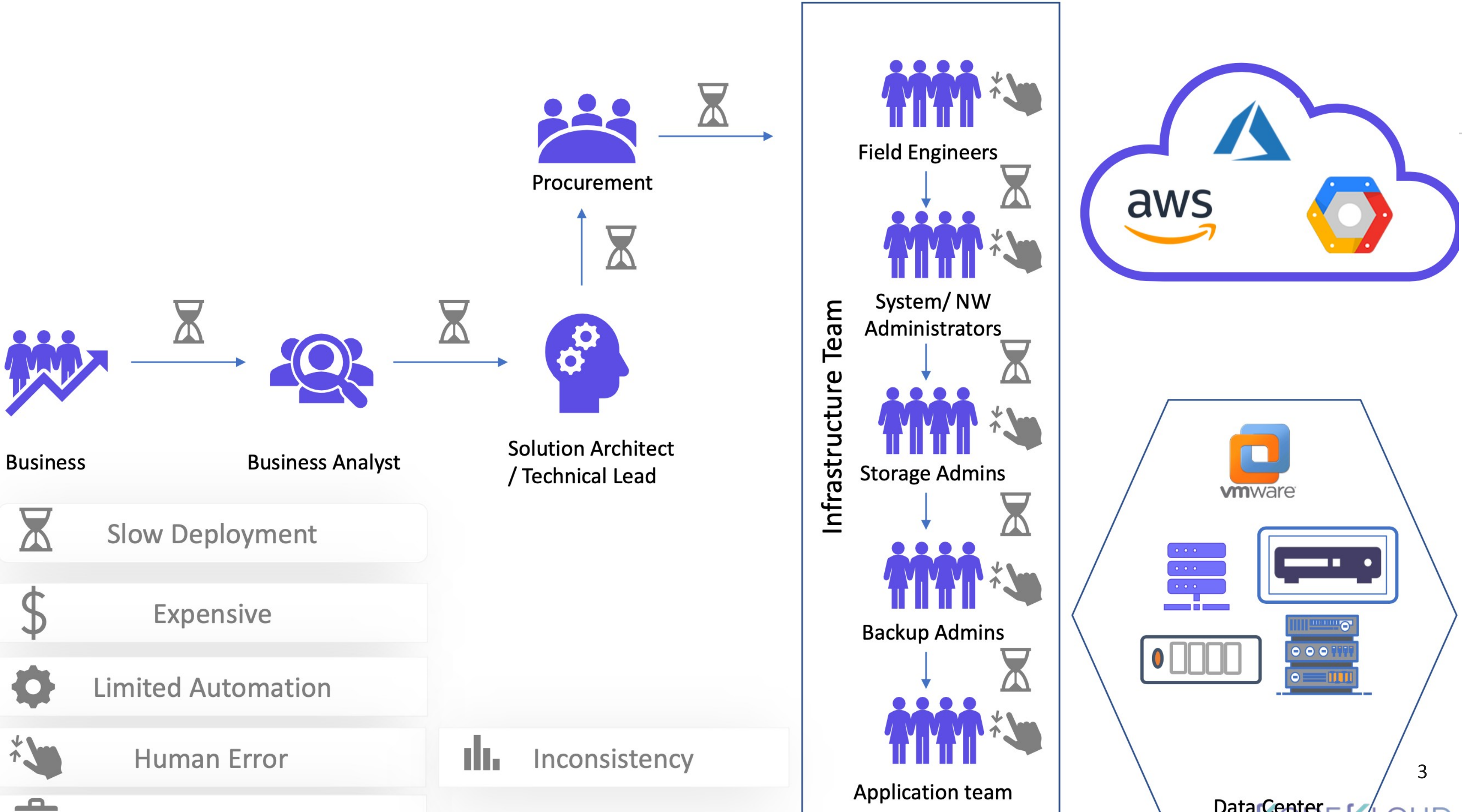
Terraform

Sabreen Salama
Linkedin

Content

- Intro to IAC
- IaC Tools
- What is terraform?
- Terraform basics
- Terraform state
- Terraform remote state
- Terraform Provisioner
- Working with terraform
- Terraform import
- Terraform modules
- CloudFormation



Business

Business Analyst

Solution Architect / Technical Lead

Procurement

Infrastructure Team

Field Engineers

System/ NW Administrators

Storage Admins

Backup Admins

Application team

Data Center

Slow Deployment

Expensive

Limited Automation

Human Error

Inconsistency

Before IAC?

- In the past, managing IT infrastructure was a hard job. System administrators had to manually manage and configure all of the hardware and software that was needed for the applications to run and this takes more time and efforts some human errors due to inconsistence

Infrastructure as code

Allow us to create and manage infrastructure with configuration files rather than through a graphical user interface or manual handling scripts These files can be versioned, reused, and shared.

Types of IAC Tools

Configuration Management

- ANSIBLE
- puppet
- SALTSTACK

Server Templating

- docker
- HashiCorp Packer
- HashiCorp Vagrant

Provisioning Tools

- HashiCorp Terraform
- CloudFormation

Why using infrastructure as code?

- Consistence: you guarantee the same configurations will be deployed over and over, without discrepancies
- Lower cost
- Speed: enables you to quickly set up your complete infrastructure by running a script
- Accountability: you can version IaC configuration files like any source code file configuration

Terraform

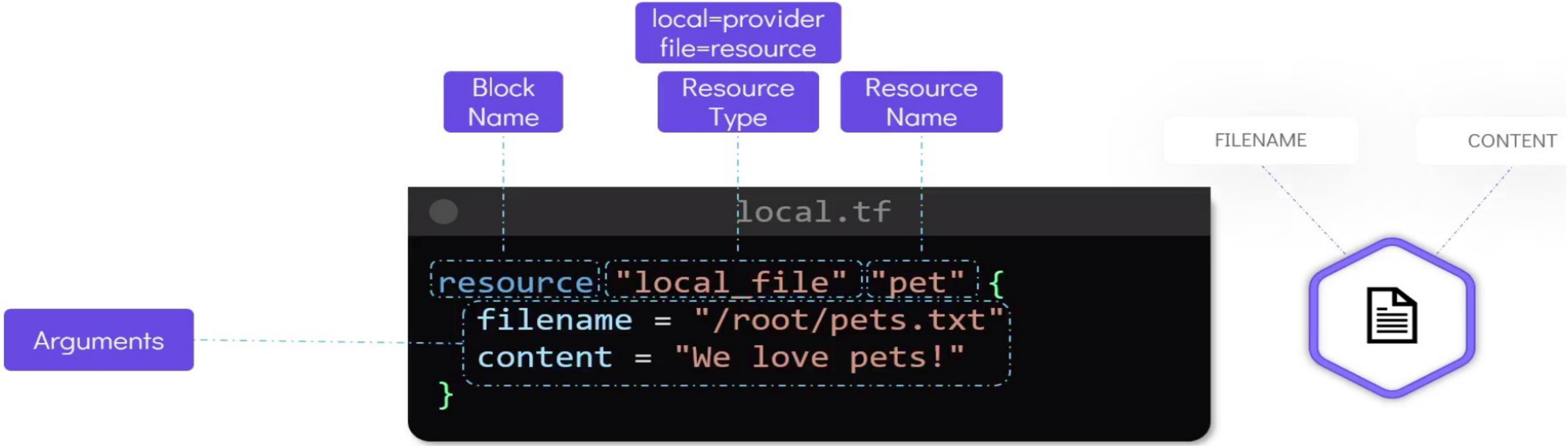
- Open source , Free , created by HashiCorp’s infrastructure as code tool . It lets you define resources and infrastructure in human-readable, declarative configuration files, and manages your infrastructure’s lifecycle to many providers, can be applied on private , public cloud , and on prem.

- Why using terraform?
  - Terraform can manage infrastructure on multiple cloud platforms
  - The human-readable configuration language helps you write infrastructure code quickly
  - Terraform's state allows you to track resource changes throughout your deployments.
  - You can commit your configurations to version control to

1

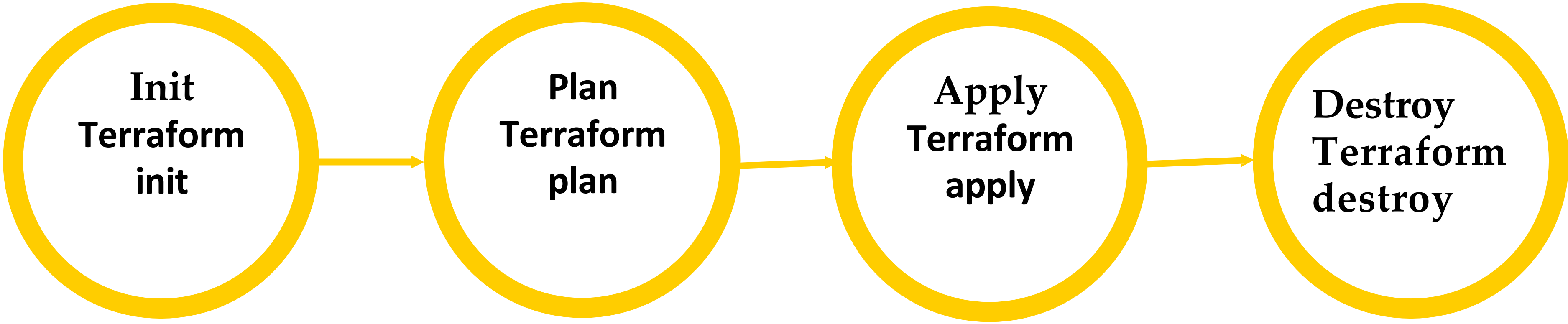
Terraform insatll

HCL



```
resource "local_file" "pet" {
  filename = "/root/pets.txt"
  content = "We love pets!"
}
```

Terraform Lifecycle



Init
Terraform
init

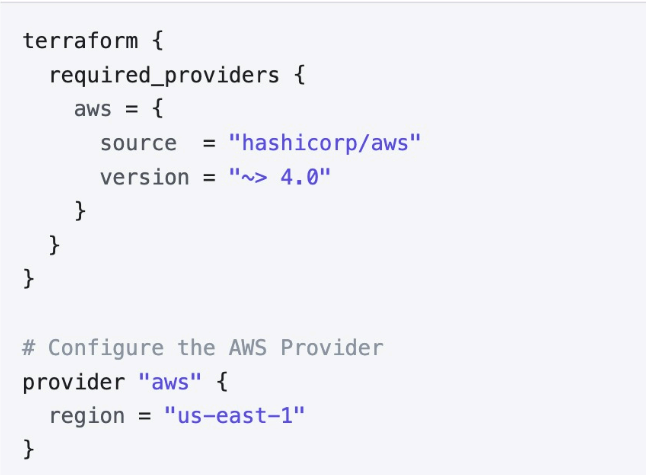
Plan
Terraform
plan

Apply
Terraform
apply

Destroy
Terraform
destroy

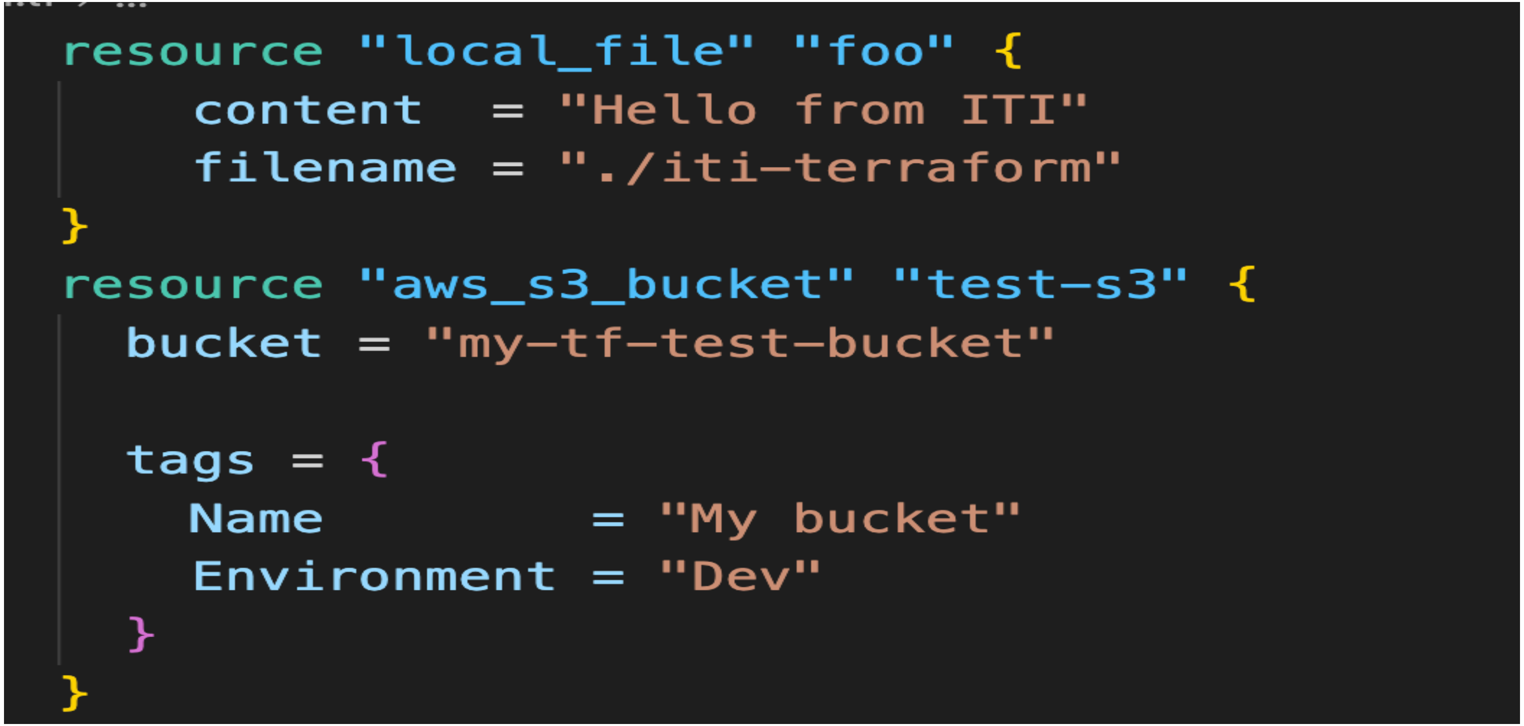
provider

◉A provider is a plugin that interacts with the various APIs required to create, update, and delete various resource



```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

# Configure the AWS Provider
provider "aws" {
  region = "us-east-1"
}
```

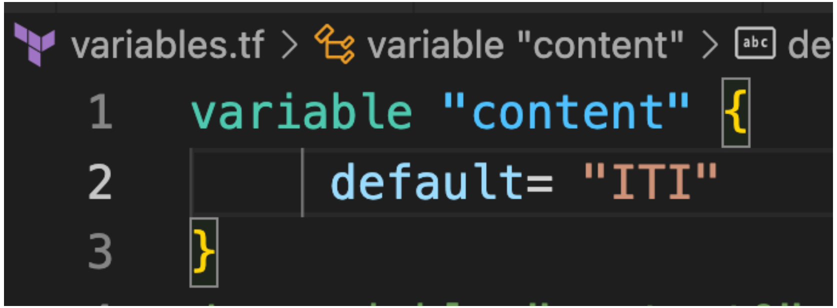
Contd: Multiple provider



```
resource "local_file" "foo" {
    content  = "Hello from ITI"
    filename = "./iti-terraform"
}
resource "aws_s3_bucket" "test-s3" {
  bucket = "my-tf-test-bucket"

  tags = {
    Name        = "My bucket"
    Environment = "Dev"
  }
}
```

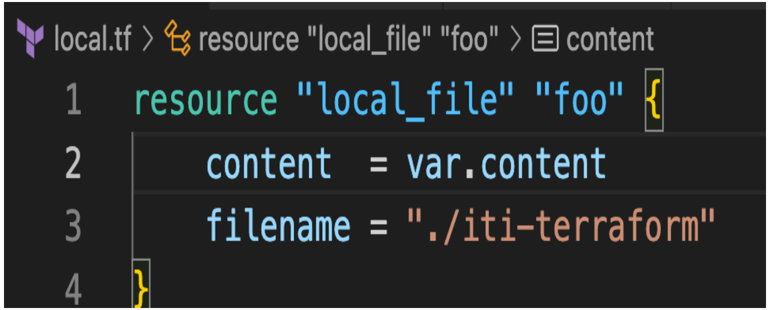
Input variables

- it is a key-value pair used as parameters to input values at run time to enable reusability to our code.



variables.tf > variable "content" > de

```
variable "content" {
    default= "ITI"
}
```

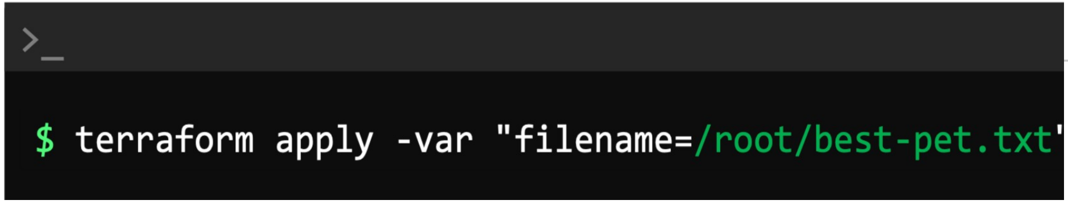


local.tf > resource "local_file" "foo" > content

```
resource "local_file" "foo" {
    content  = var.content
    filename = "./iti-terraform"
}
```

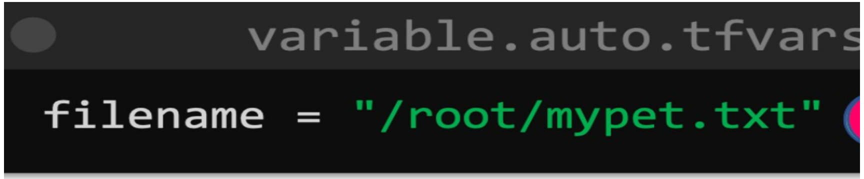
Variable Definition Precedence

1



```
$ terraform apply -var "filename=/root/best-pet.txt'
```

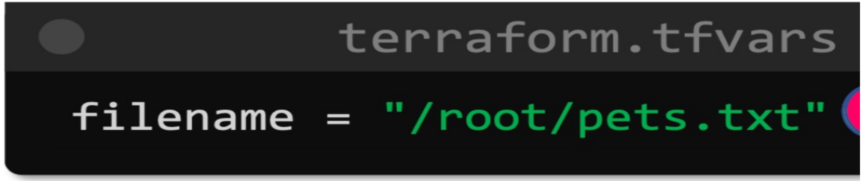
2



variable.auto.tfvars

```
filename = "/root/mypet.txt"
```

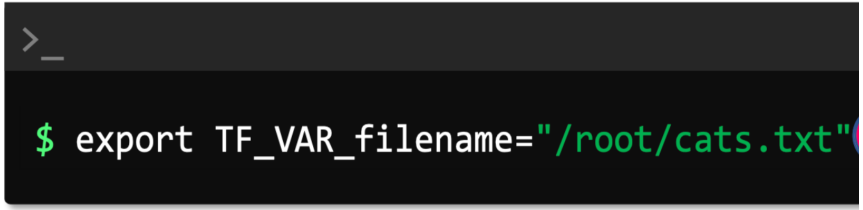
3



terraform.tfvars

```
filename = "/root/pets.txt"
```

4

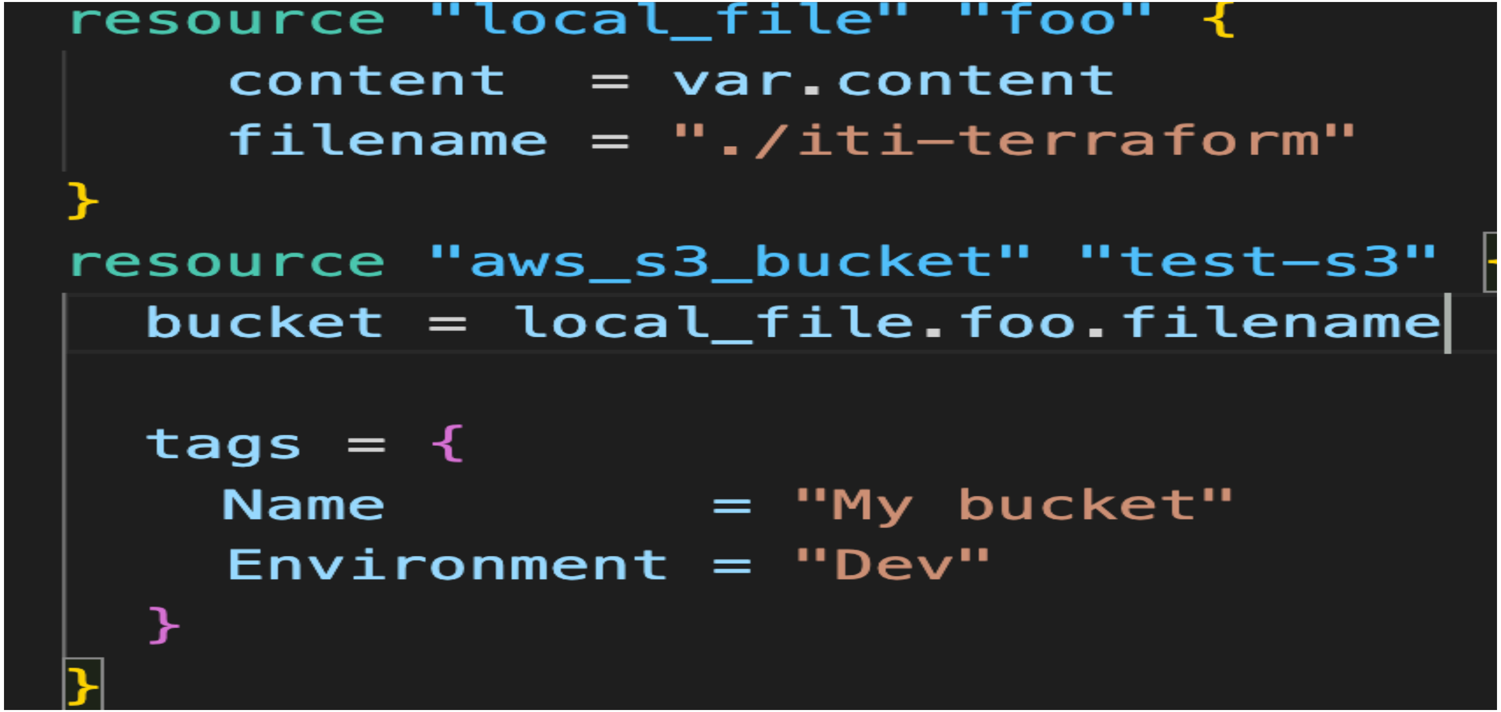


```
$ export TF_VAR_filename="/root/cats.txt"
```

Dependency of resources

- Resource attribute or reference attribute: to use output of a resource as an input for another resource

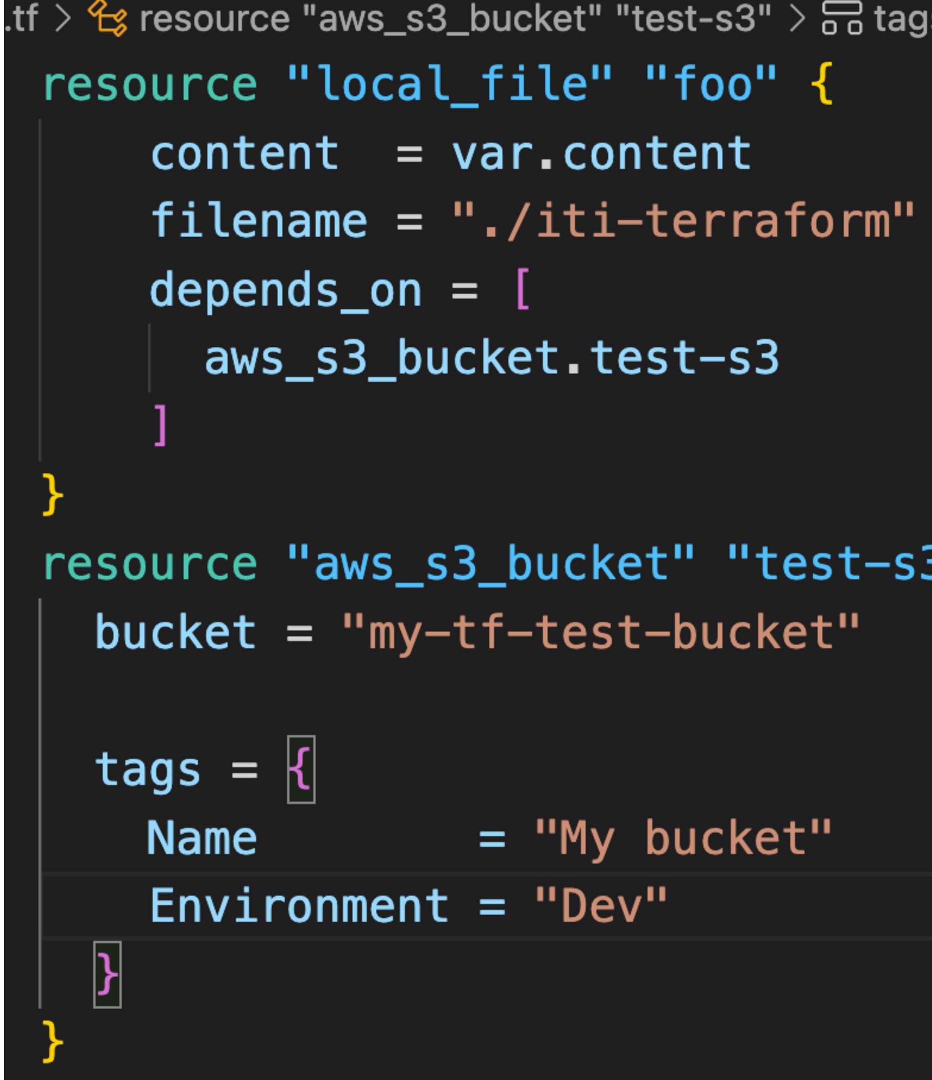
Reference Attributes



```
resource "local_file" "foo" {
    content  = var.content
    filename = "./iti-terraform"
}
resource "aws_s3_bucket" "test-s3" {
  bucket = local_file.foo.filename

  tags = {
    Name        = "My bucket"
    Environment = "Dev"
  }
}
```

Contd Dependency of resources

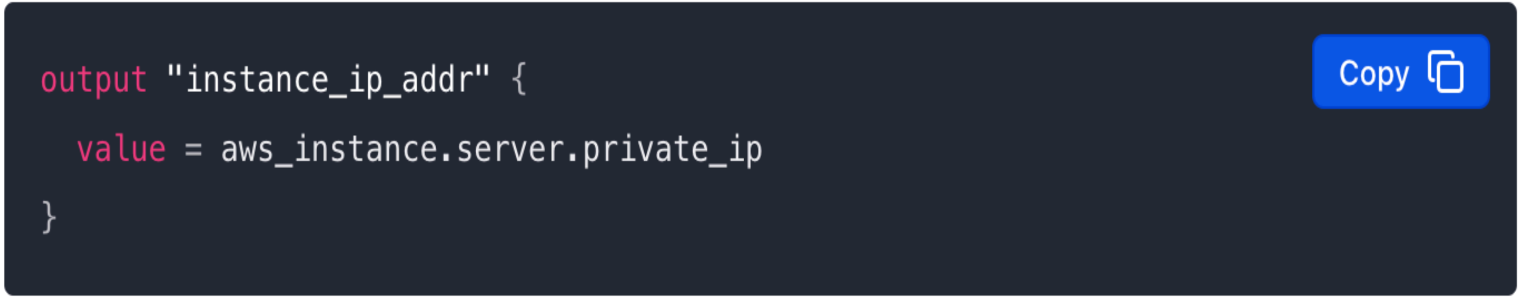
- Explict dependency: Make use of all configuration files without making use of reference



```
resource "local_file" "foo" {
    content  = var.content
    filename = "./iti-terraform"
    depends_on = [
      aws_s3_bucket.test-s3
    ]
}
resource "aws_s3_bucket" "test-s3
  bucket = "my-tf-test-bucket"

  tags = {
    Name        = "My bucket"
    Environment = "Dev"
  }
}
```

Outputs

- Return attributes for a resource to print it to the screen or to fed it to configuration tool as ansible



```
output "instance_ip_addr" {
  value = aws_instance.server.private_ip
}
```

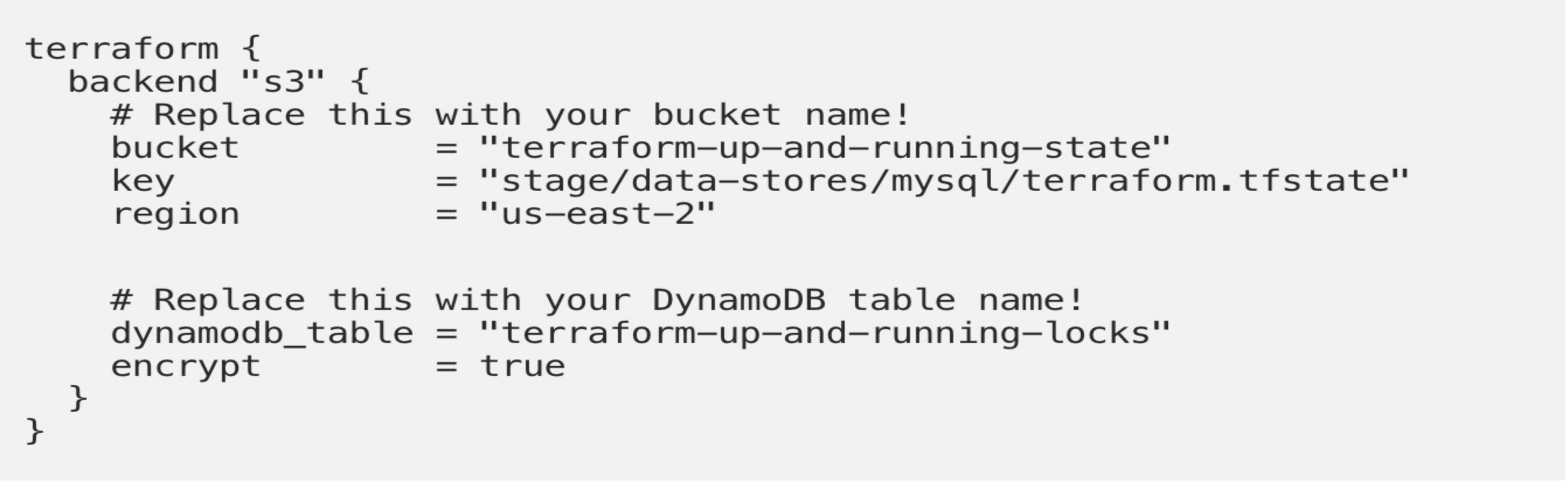
Terraform state

It is a json data structure file that maps a real world infra to the resource definition in configuration files

Purpose of state file:

- Mapping resources into real world resources
- Tracking metadata such as dependency
- Collaboration

Terraform remote state



```
terraform {
  backend "s3" {
    # Replace this with your bucket name!
    bucket         = "terraform-up-and-running-state"
    key            = "stage/data-stores/mysql/terraform.tfstate"
    region         = "us-east-2"

    # Replace this with your DynamoDB table name!
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt        = true
  }
}
```

Terraform commands

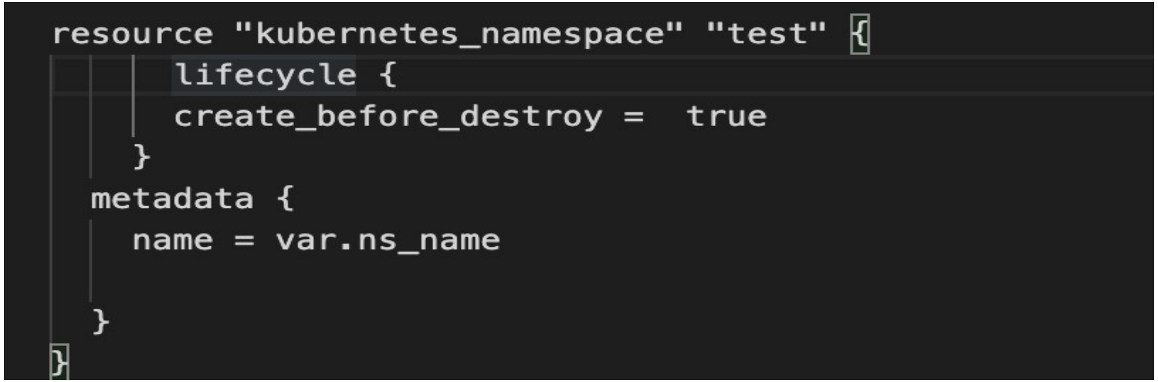
- Terraform validate
- Terraform fmt
- Terraform show
- Terraform providers
- Terraform output
- Terraform refresh
- Terraform graph

Terraform commands

- Terraform state list
- Terraform state mv
- Terraform state rm
- Terraform state show

Lifecycle rules

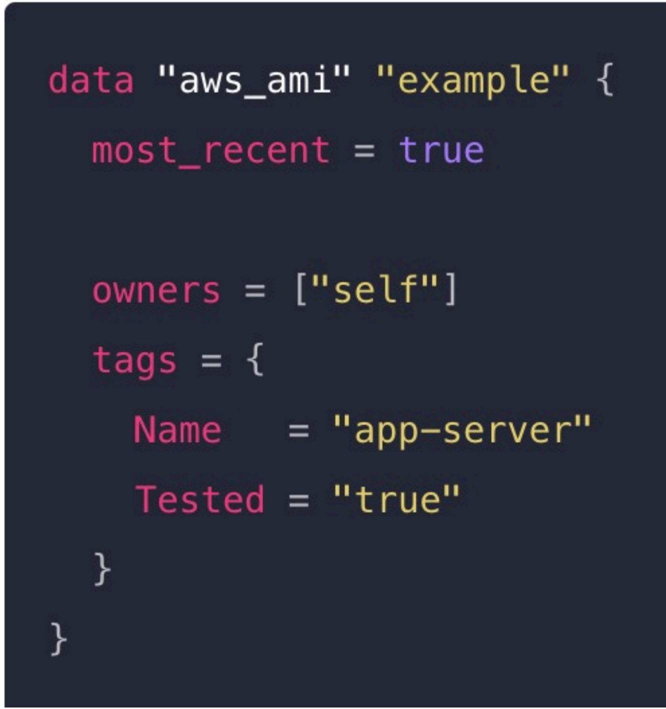
The default when we run terraform apply the old resource deleted and then the new one will be created what about not deleting the old ones or creating first then deleting



```
resource "kubernetes_namespace" "test" {
      lifecycle {
      create_before_destroy =  true
    }
  metadata {
    name = var.ns_name

  }
}
```

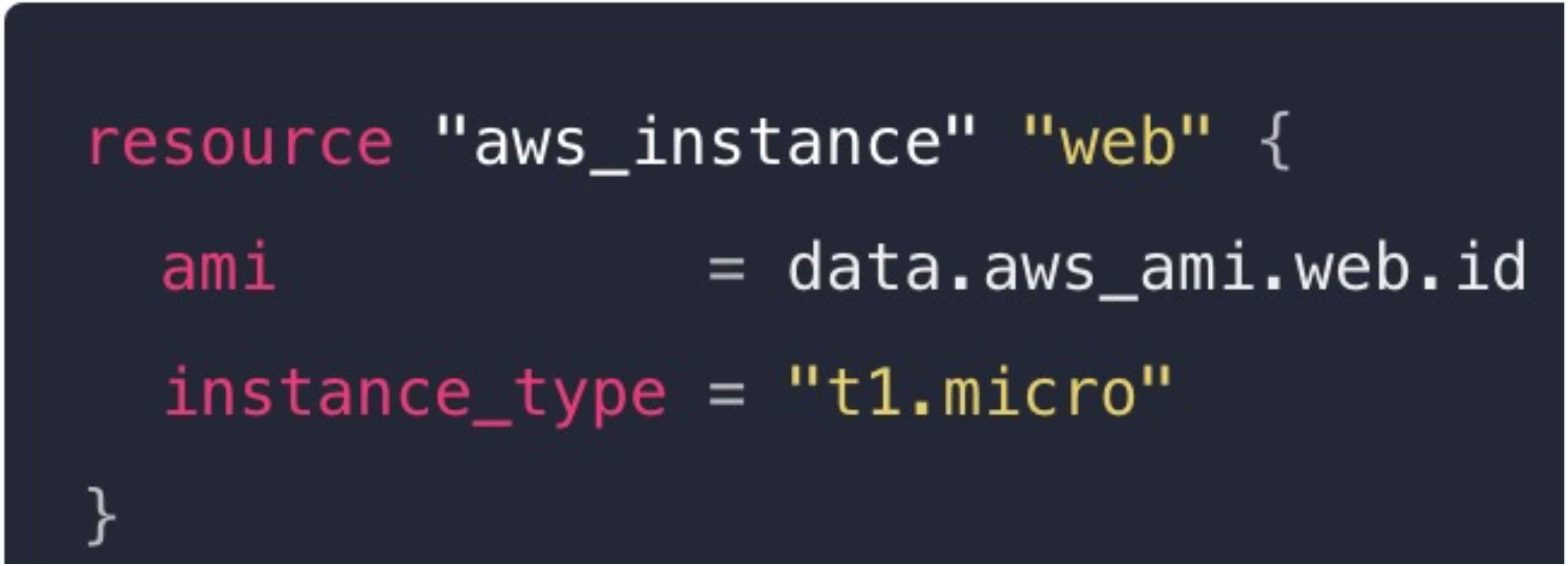
Datasources

Allow Terraform to read attributes from a resources that were created manually outside of terraform



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

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

Resource VS DataSource

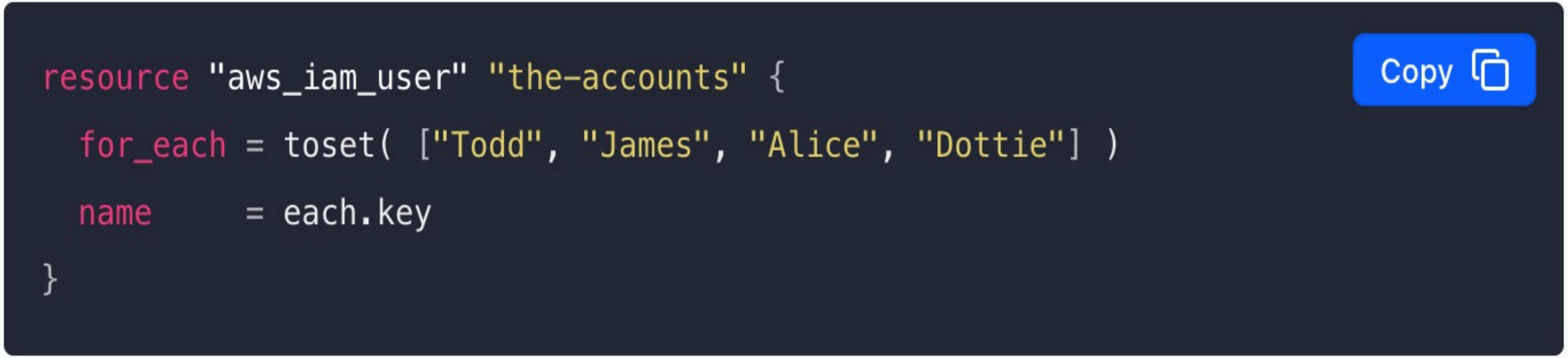


```
resource "aws_instance" "web" {
  ami           = data.aws_ami.web.id
  instance_type = "t1.micro"
}
```

Meta argument

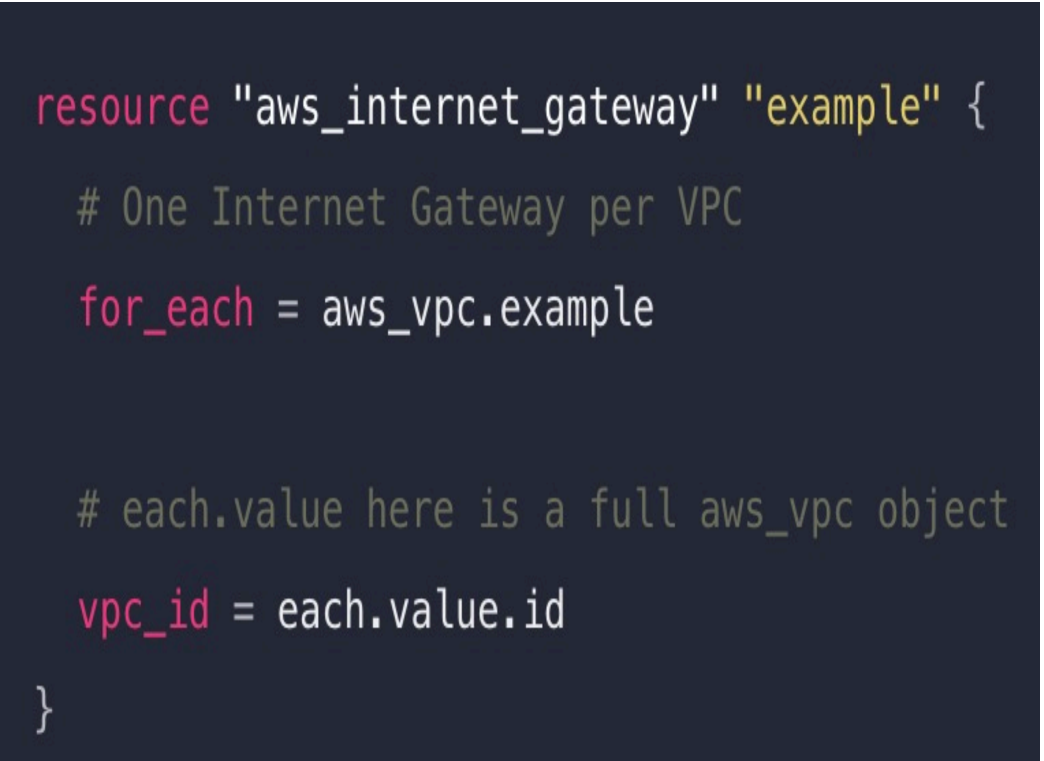
To change the behaviour of a resource as depends_on and lifecycle

Now we will deal with count and for_each

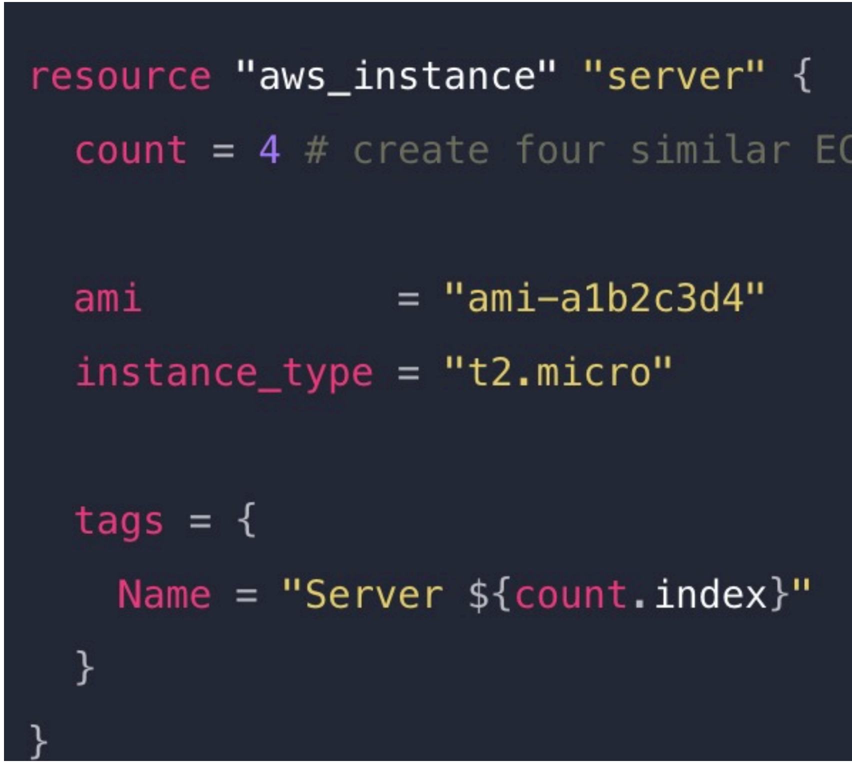


```
resource "aws_iam_user" "the-accounts" {
  for_each = toset( ["Todd", "James", "Alice", "Dottie"] )
  name     = each.key
}
```

Count VS for_each

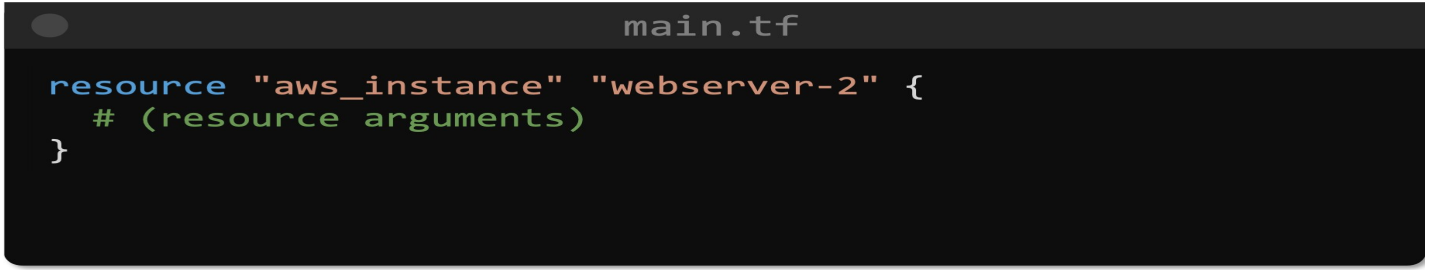


```
resource "aws_internet_gateway" "example" {
  # One Internet Gateway per VPC
  for_each = aws_vpc.example

  # each.value here is a full aws_vpc object
  vpc_id = each.value.id
}
```

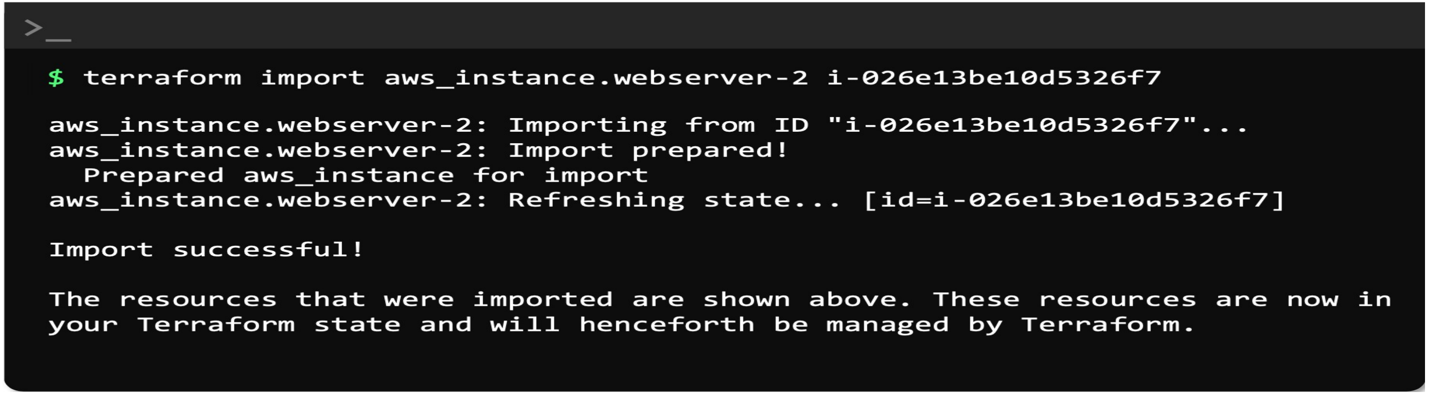


```
resource "aws_instance" "server" {
  count = 4 # create four similar EC

  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"

  tags = {
    Name = "Server ${count.index}"
  }
}
```

Terraform import



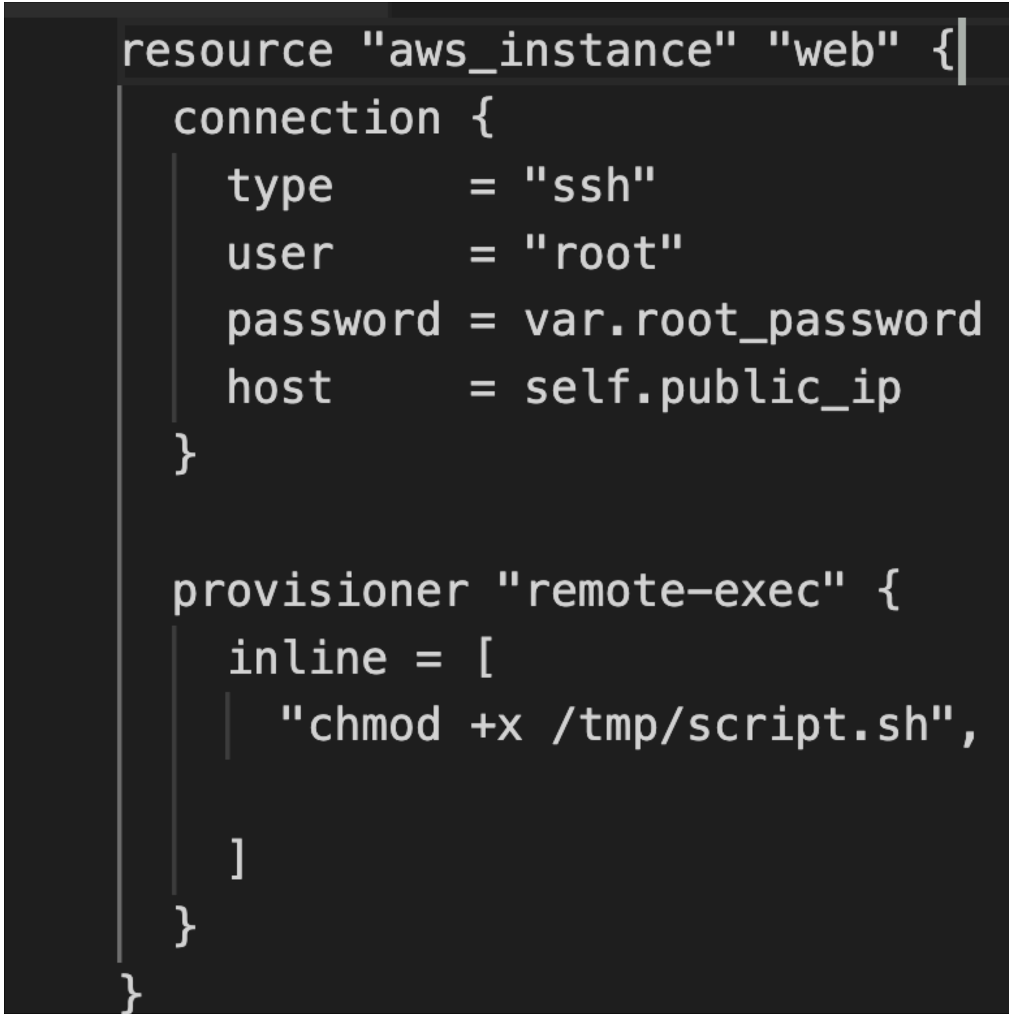
```
resource "aws_instance" "webserver-2" {
  # (resource arguments)
}
```



```
$ terraform import aws_instance.webserver-2 i-026e13be10d5326f7

aws_instance.webserver-2: Importing from ID "i-026e13be10d5326f7"...
aws_instance.webserver-2: Import prepared!
  Prepared aws_instance for import
aws_instance.webserver-2: Refreshing state... [id=i-026e13be10d5326f7]

Import successful!

The resources that were imported are shown above. These resources are now in
your Terraform state and will henceforth be managed by Terraform.
```

remote-exec Provisioner

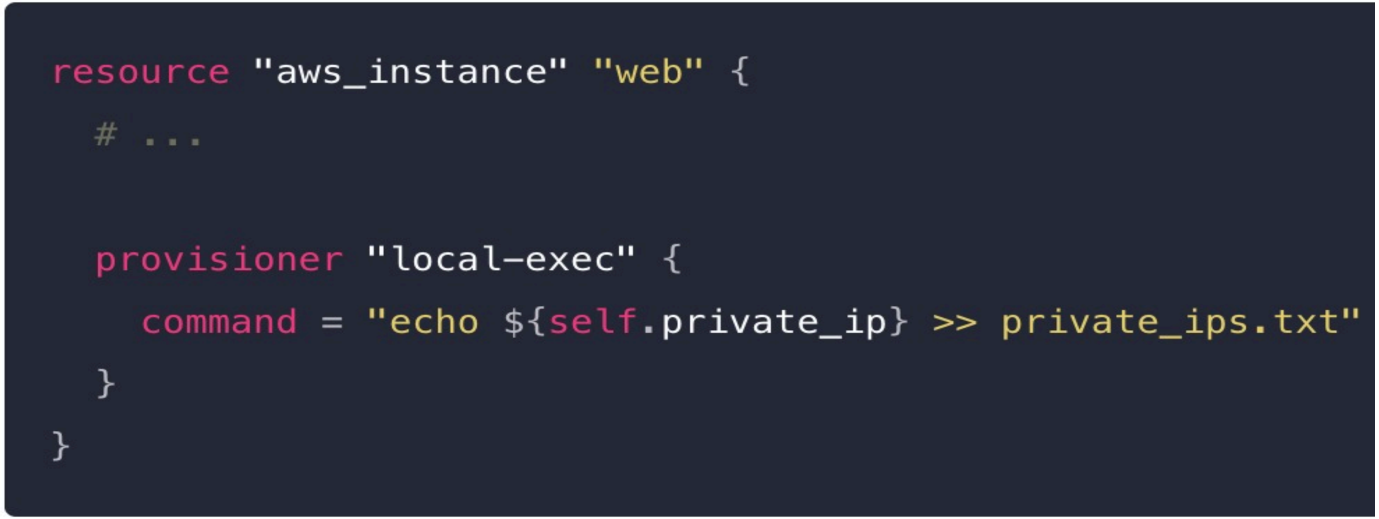
The remote-exec provisioner invokes a script on a remote resource after it is created. This can be used to run a configuration management tool, bootstrap into a cluster, etc



```
resource "aws_instance" "web" {
  connection {
    type     = "ssh"
    user     = "root"
    password = var.root_password
    host     = self.public_ip
  }

  provisioner "remote-exec" {
    inline = [
      "chmod +x /tmp/script.sh",

    ]
  }
}
```

local-exec Provisioner

The local-exec provisioner invokes a local executable after a resource is created. This invokes a process on the machine running Terraform, not on the resource.



```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo ${self.private_ip} >> private_ips.txt"
  }
}
```

Terraform modules

A module is a container for multiple resources that are used together. Modules can be used to create lightweight abstractions, so that you can describe your infrastructure in terms of its architecture, rather than directly in terms of physical objects.

child module "server"

```
resource "type" "vm" {
  network_id = var.network_id
  . . .
  . . .
}


variable "network_id" {
}
```

child module "network"

```
resource "type" "network" {
  argument = value
  argument = value
  argument = value
  . . .
}

output "network_id" {
  value = type.network.id
}
```

I have a solution!

root module

Terraform Workspace

Each Terraform configuration has an associated backend that defines how Terraform executes operations and where Terraform stores persistent data, like state.

The persistent data stored in the backend belongs to a workspace. The backend initially has only one workspace containing one Terraform state associated with that configuration.

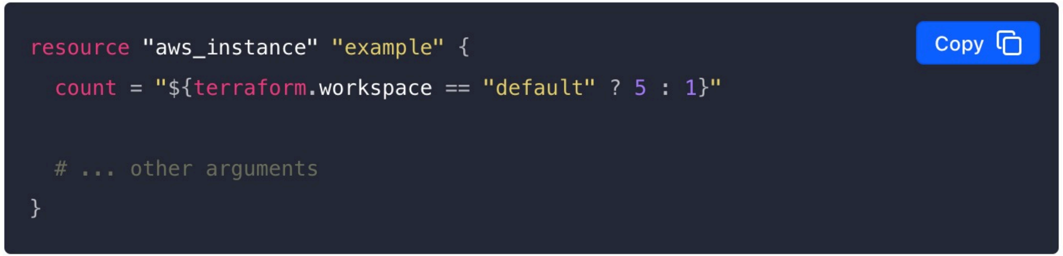
Terraform Workspace

To list all spaces:

terraform workspace list

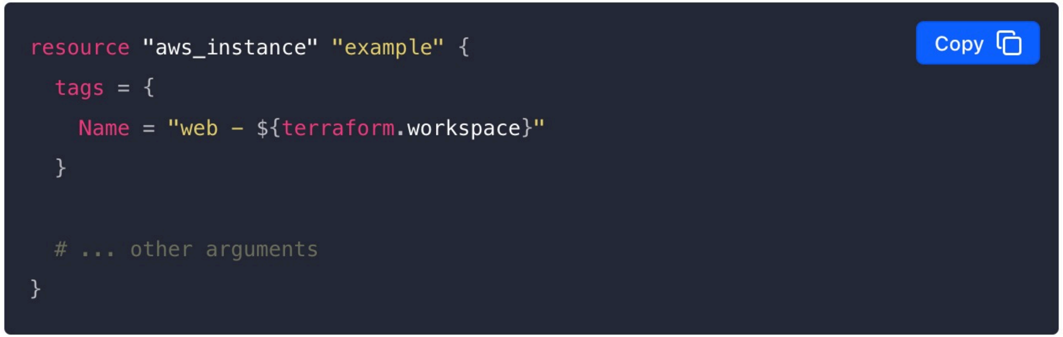
For creating new space:

terraform workspace new <space-name>



```
resource "aws_instance" "example" {
  count = "${terraform.workspace == "default" ? 5 : 1}"

  # ... other arguments
}
```

Another popular use case is using the workspace name as part of naming or tagging behavior:



```
resource "aws_instance" "example" {
  tags = {
    Name = "web - ${terraform.workspace}"
  }

  # ... other arguments
}
```

Thanks!

Any questions ?