Seshagiri Sriram

Terraform

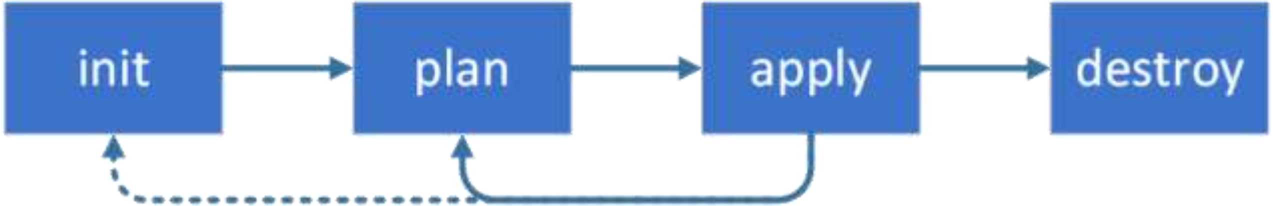
What is Terraform

Terraform is an open source infrastructure as code tool. Terraform was developed by HashiCorp company who is based in San Francisco, CA. We can also say that Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently.

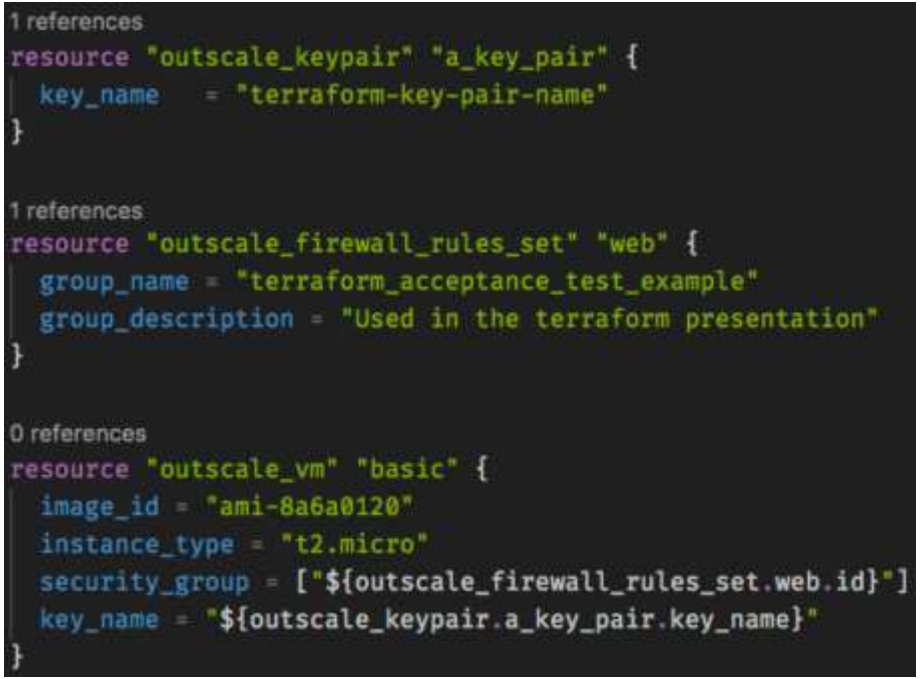
Terraform principles

Terraform has these three principal simple steps:

- Init
- Plan
- Apply



Terraform sample code



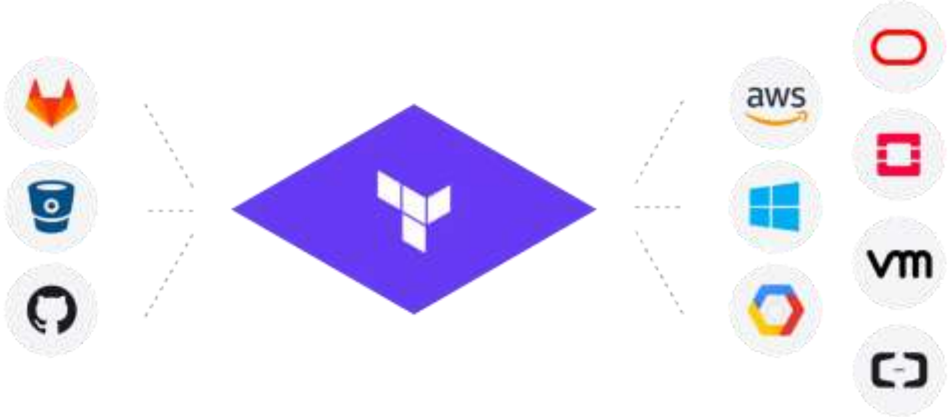
```
1 references
resource "outscale_keypair" "a_key_pair" {
  key_name   = "terraform-key-pair-name"
}

1 references
resource "outscale_firewall_rules_set" "web" {
  group_name = "terraform_acceptance_test_example"
  group_description = "Used in the terraform presentation"
}

0 references
resource "outscale_vm" "basic" {
  image_id = "ami-8a6a0120"
  instance_type = "t2.micro"
  security_group = ["${outscale_firewall_rules_set.web.id}"]
  key_name = "${outscale_keypair.a_key_pair.key_name}"
}
```

Multi-cloud

Terraform can be used to manage multi-cloud It can combine multiple provides in a single workflow which is a very nice feature. Terraform provides one consistent workflow for developers and operators to provision resources on any infrastructure provider. One workflow to learn increases user productivity, and also reduces organizational risk as that becomes one workflow to secure, one workflow to audit, and one workflow to govern.

Terraform providers

- Right now, there are more than 100 providers and those individually can manage over a thousand resources. Providers are responsible to provide API interaction. You can find the list of providers on the terraform website.



Code samples:

github.com/gruntwork-io/infrastructure-as-code-training

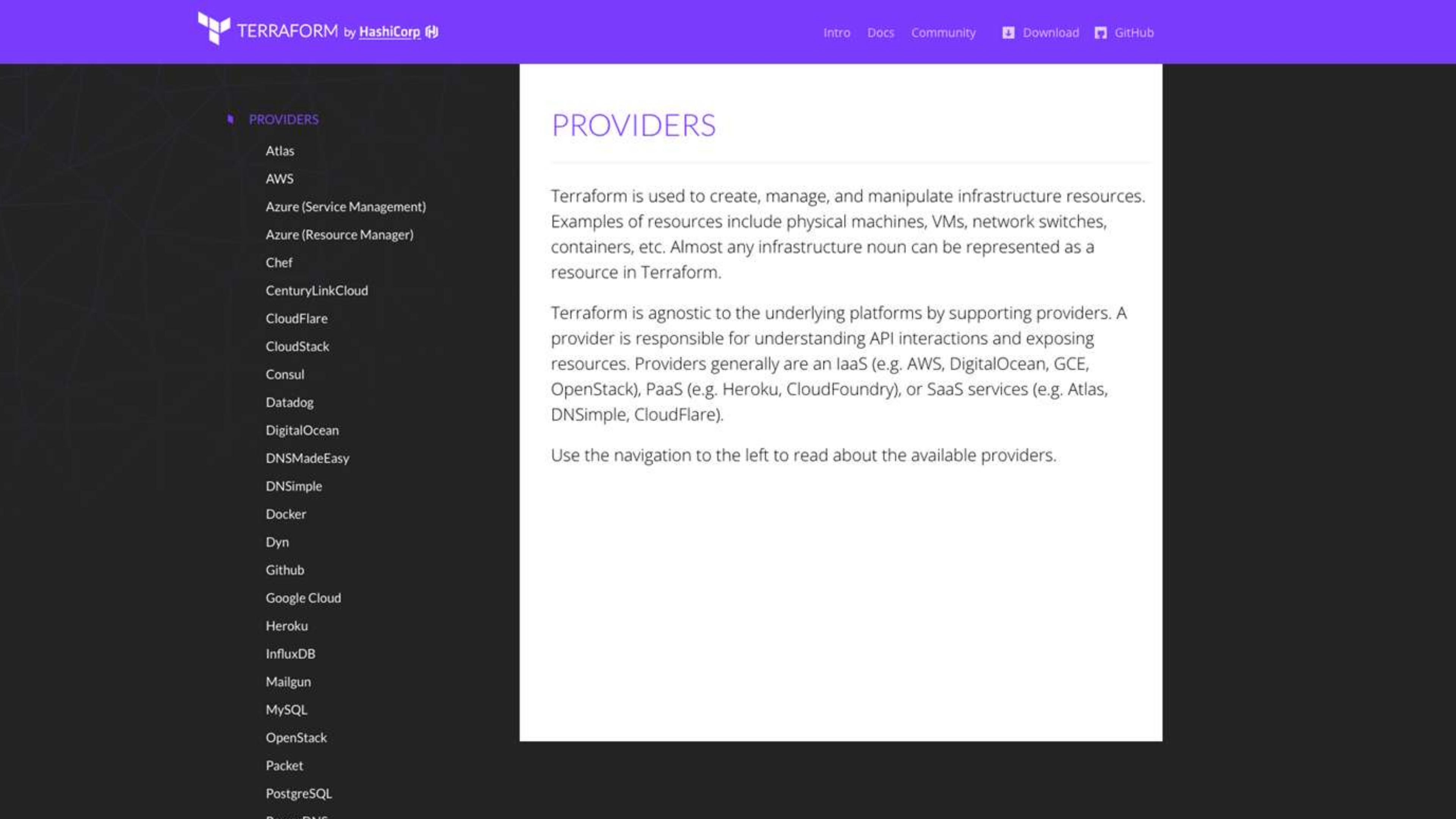
Getting started

Terraform is a tool for provisioning infrastructure

It supports many providers (cloud agnostic)

And many resources for each provider

You define resources as code in Terraform templates



TERRAFORM by HashiCorp

Intro Docs Community Download GitHub

PROVIDERS

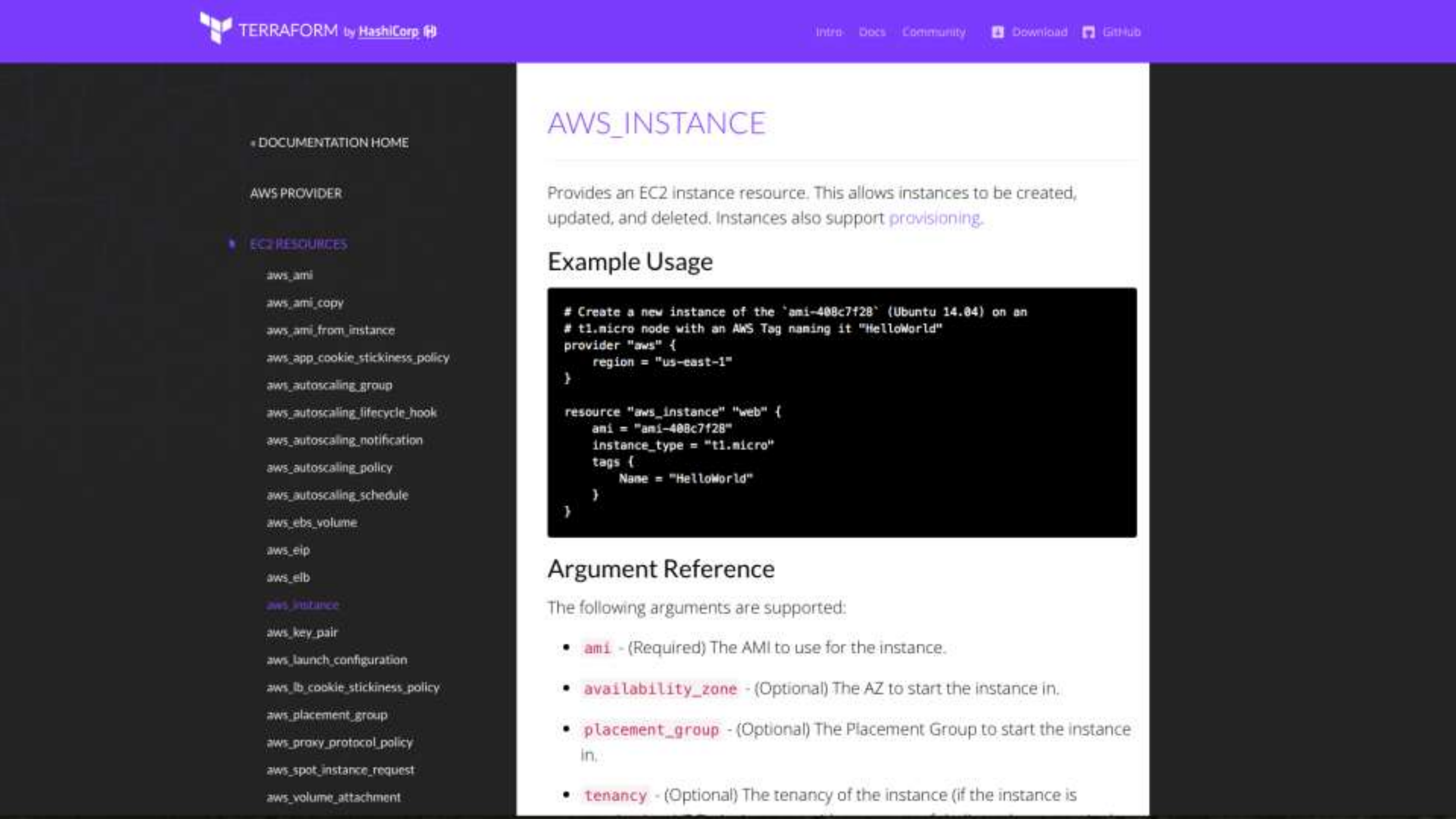
- Atlas
- AWS
- Azure (Service Management)
- Azure (Resource Manager)
- Chef
- CenturyLinkCloud
- CloudFlare
- CloudStack
- Consul
- Datadog
- DigitalOcean
- DNSMadeEasy
- DNSimple
- Docker
- Dyn
- Github
- Google Cloud
- Heroku
- InfluxDB
- Mailgun
- MySQL
- OpenStack
- Packet
- PostgreSQL

PROVIDERS

Terraform is used to create, manage, and manipulate infrastructure resources. Examples of resources include physical machines, VMs, network switches, containers, etc. Almost any infrastructure noun can be represented as a resource in Terraform.

Terraform is agnostic to the underlying platforms by supporting providers. A provider is responsible for understanding API interactions and exposing resources. Providers generally are an IaaS (e.g. AWS, DigitalOcean, GCE, OpenStack), PaaS (e.g. Heroku, CloudFoundry), or SaaS services (e.g. Atlas, DNSimple, CloudFlare).

Use the navigation to the left to read about the available providers.



AWS_INSTANCE

Provides an EC2 instance resource. This allows instances to be created, updated, and deleted. Instances also support provisioning.

Example Usage

```
# Create a new instance of the `ami-408c7f28` (Ubuntu 14.04) on an
# t1.micro node with an AWS Tag naming it "HelloWorld"
provider "aws" {
    region = "us-east-1"
}

resource "aws_instance" "web" {
    ami = "ami-408c7f28"
    instance_type = "t1.micro"
    tags {
        Name = "HelloWorld"
    }
}
```

Argument Reference

The following arguments are supported:

- `ami` - (Required) The AMI to use for the instance.
- `availability_zone` - (Optional) The AZ to start the instance in.
- `placement_group` - (Optional) The Placement Group to start the instance in.
- `tenancy` - (Optional) The tenancy of the instance (if the instance is

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

resource "aws_instance" "example" {
    ami = "ami-408c7f28"
    instance_type = "t2.micro"
    tags { Name = "terraform-example" }
}
```

This template creates a single EC2 instance in AWS

```
> terraform plan
+ aws_instance.example
    ami:                ""  => "ami-408c7f28"
    instance_type:      ""  => "t2.micro"
    key_name:           ""  => "<computed>"
    private_ip:         ""  => "<computed>"
    public_ip:          ""  => "<computed>"


Plan: 1 to add, 0 to change, 0 to destroy.
```

Use the plan command to see what you’re about to deploy

```
> terraform apply
aws_instance.example: Creating...
    ami:               "" => "ami-408c7f28"
    instance_type:     "" => "t2.micro"
    key_name:          "" => "<computed>"
    private_ip:        "" => "<computed>"
    public_ip:         "" => "<computed>”
aws_instance.example: Creation complete

Apply complete! Resources: 1 added, 0 changed, 0
destroyed.
```

Use the apply command to apply the changes

Terraform

Templates can be parameterized using Variables

```
variable "name" {
    description = "The name of the EC2 instance"
}
```

Description, default and types are optional

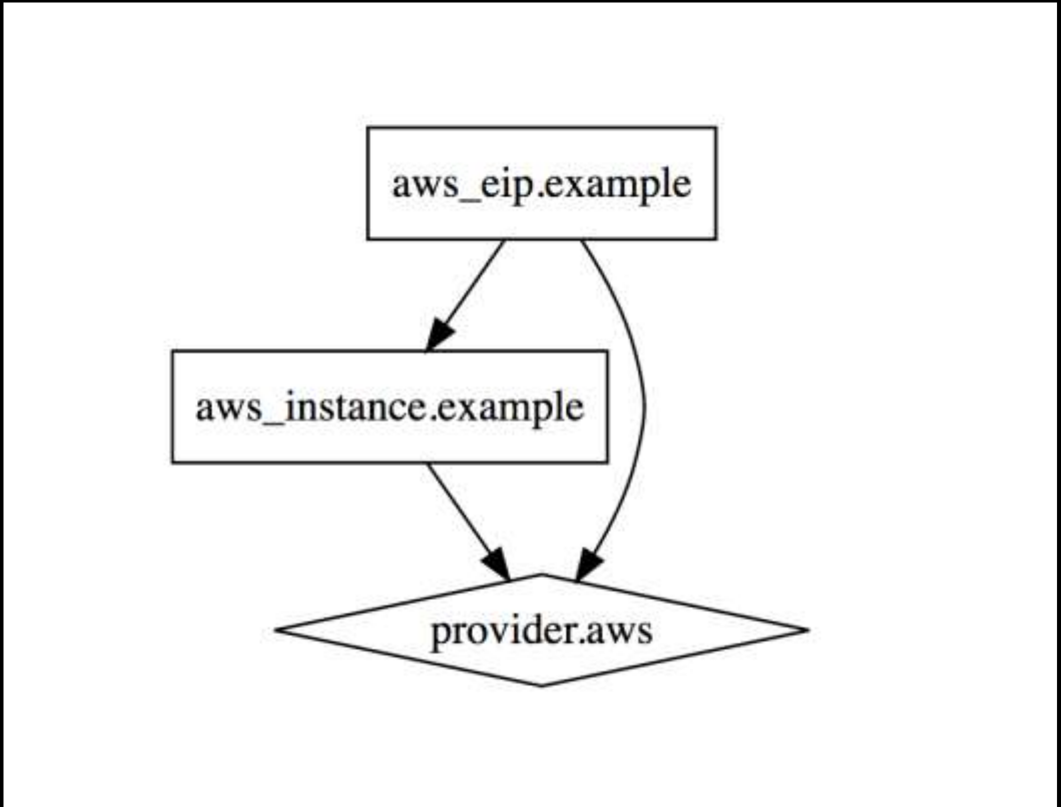
Terraform

```
variable "name" {
    description = "The name of the EC2 instance"
}
resource "aws_instance" "example" {
    ami = "ami-408c7f28"
    instance_type = "t2.micro"
    tags { Name = "${var.name}" }
}
```

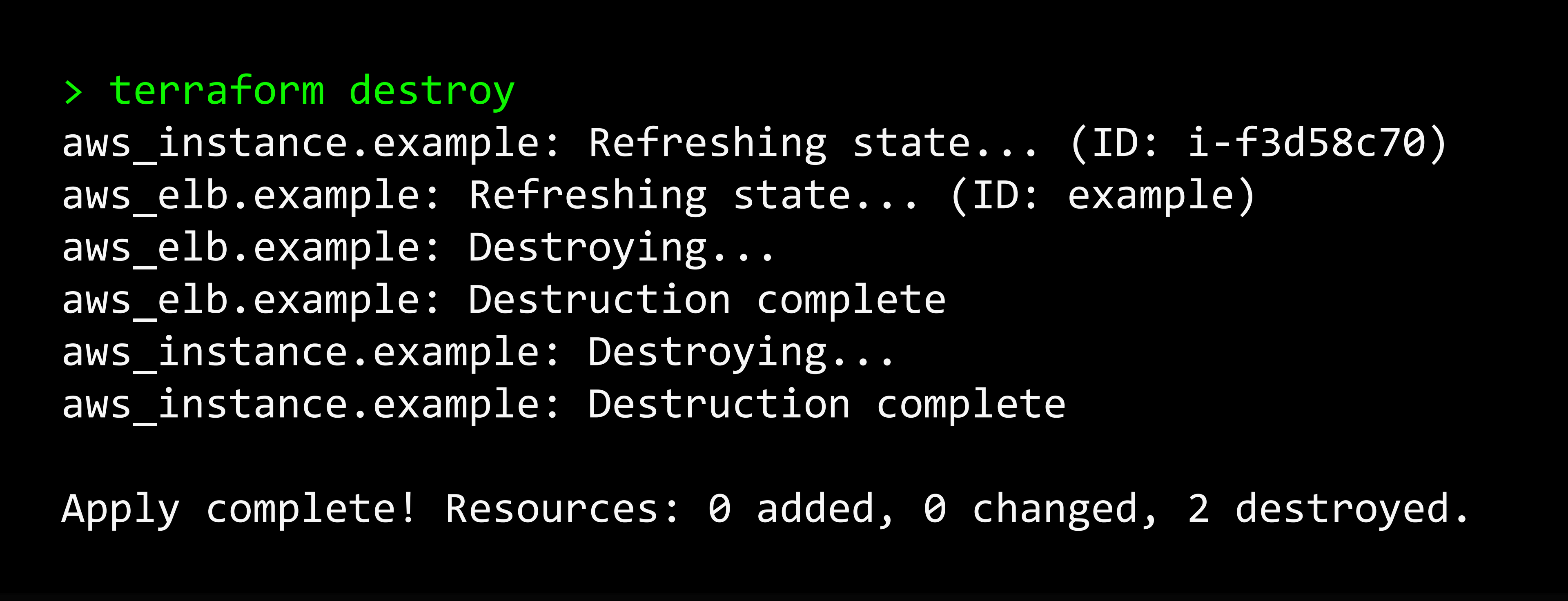
Terraform will prompt for a variable value when plan is done.
Variables can also be passed with –var <varname>=<varvalue>

```
resource "aws_eip" "example" {
    instance = "${aws_instance.example.id}”
}

resource "aws_instance" "example" {
    ami = "ami-408c7f28"
    instance_type = "t2.micro"
    tags { Name = "${var.name}" }
}
```

Notice the use of `${}` to depend on the `id` of the `aws_instance`



Terraform automatically builds a dependency graph



```
> terraform destroy
aws_instance.example: Refreshing state... (ID: i-f3d58c70)
aws_elb.example: Refreshing state... (ID: example)
aws_elb.example: Destroying...
aws_elb.example: Destruction complete
aws_instance.example: Destroying...
aws_instance.example: Destruction complete

Apply complete! Resources: 0 added, 0 changed, 2 destroyed.
```

Use the `destroy` command to clean up

Terraform

Terraform maintains state.

By default, stored in .tfstate files locally.

Preferred to store state remotely in S3, Atlas, Consul etc.

```
> terraform remote config \
  -backend=s3 \
  -backend-config=bucket=my-s3-bucket \
  -backend-config=key=terraform.tfstate
  -backend-config=encrypt=true \
  -backend-config=region=us-east-1
```

You can enable remote state storage in S3, Atlas, Consul, etc.

Terraform

Atlas is expensive

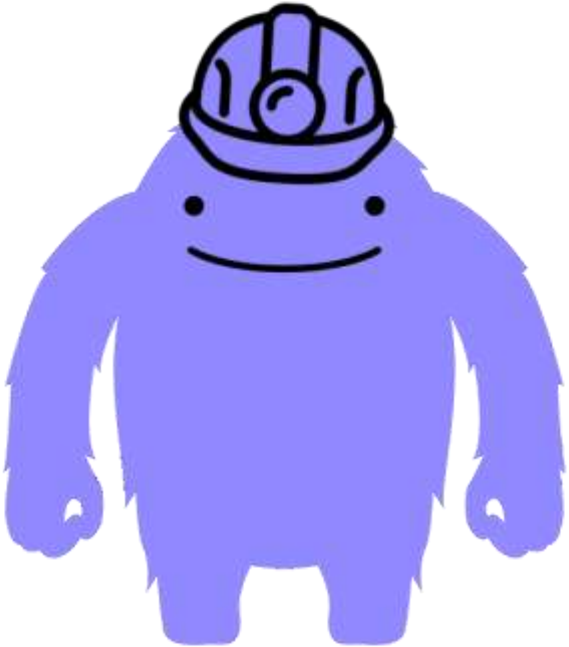
An alternative is Terragrunt

Terragrunt is an open source wrapper for Terraform

Uses Dynamodb and looks for a .terragrunt file

An example

```
terragrunt plan == terraform plan
```



Terraform Modules

A module is similar to a blueprint

In plain words, it is a folder with terraform templates

A normal convention is to

vars.tf	Keep Variables and module inputs here
main.tf	Main module
outputs.tf	Keep outputs from application here

Terraform Modules

Terraform Modules

Source specifies the folder where the module is located.

Can be re-used any number of times

It can even refer to a versioned GIT URL

```
module "example_rails_app" {
    source = "./rails-module"
    name = “Example Rails App”  # module variables
    ami = “ami-123456”          # Module variables
}
```

Terraform Modules

When using GIT modules, run the ***terraform get –update*** command.

Terraform Best Practices

01. Plan before apply

02. Stage before moving to Production

3. Isolated environments

```
stage
prod
mgmt
```

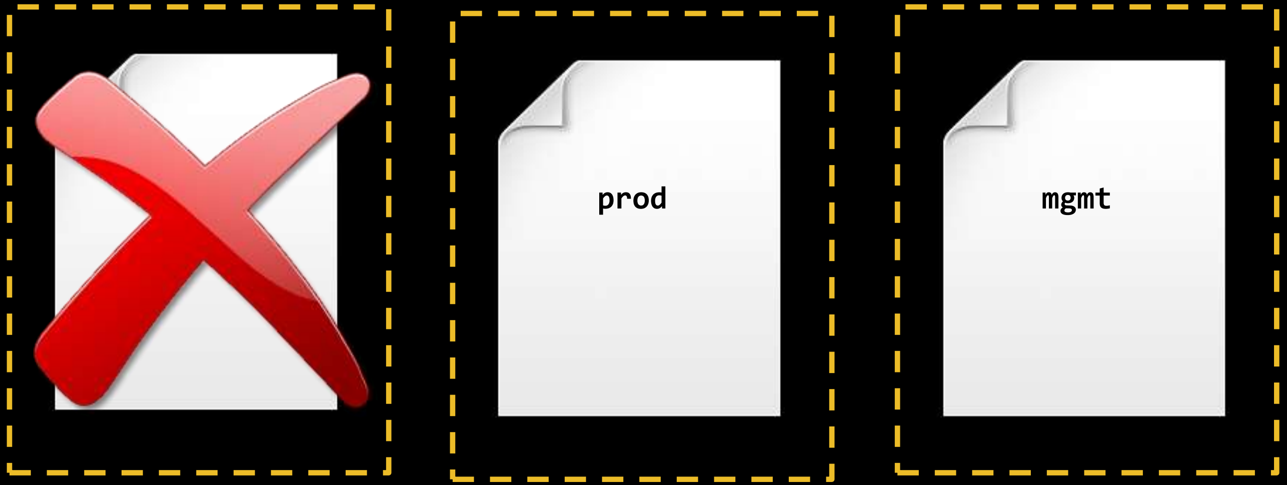
It’s tempting to define everything in 1 template

stage

prod

mgmt

What you really want is **isolation** for each environment



That way, a problem in stage doesn’t affect prod

Recommended folder structure (simplified):

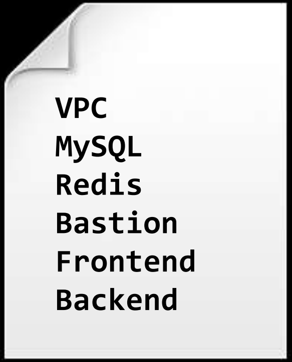
```
global (Global resources such as IAM, SNS, S3)
  └ main.tf
  └ .terragrunt

stage (Non-production workloads, testing)
  └ main.tf
  └ .terragrunt

prod (Production workloads, user-facing apps)
  └ main.tf
  └ .terragrunt

mgmt (DevOps tooling such as Jenkins, Bastion
Host)
  └ main.tf
  └ .terragrunt
```

Each folder gets its own `.tfstate`

```
global (Global resources such as IAM, SNS, S3)
  └ main.tf
  └ .terragrunt

stage (Non-production workloads, testing)
  └ main.tf
  └ .terragrunt

prod (Production workloads, user-facing apps)
  └ main.tf
  └ .terragrunt

mgmt (DevOps tooling such as Jenkins, Bastion
Host)
  └ main.tf
  └ .terragrunt
```

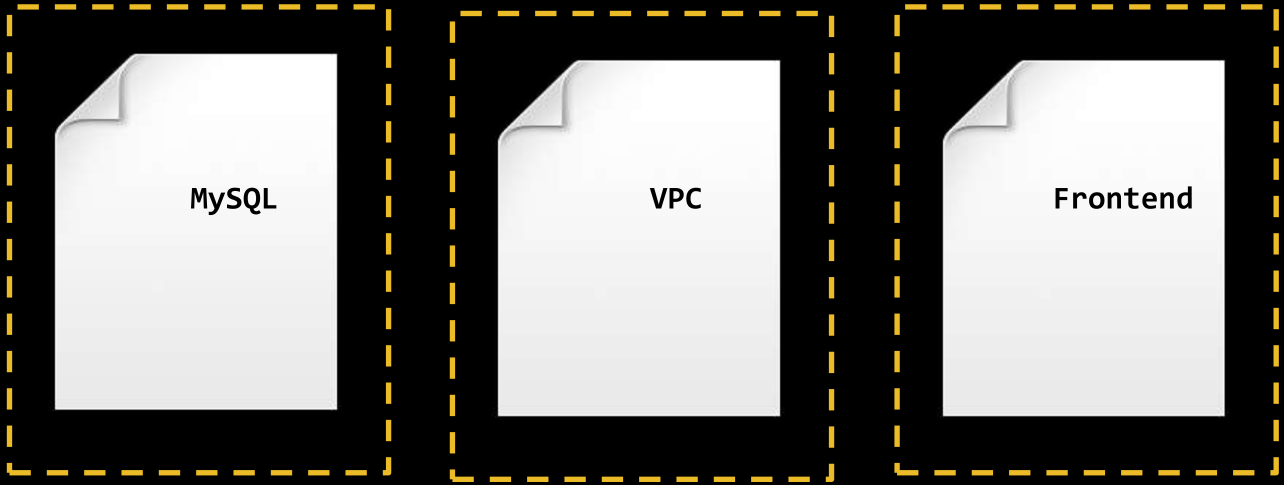
Use terraform_remote_state to share state between them

4. Isolated components



VPC
MySQL
Redis
Bastion
Frontend
Backend

It’s tempting to define everything in 1 template for all components.



MySQL

VPC

Frontend

What you really want is **isolation** for each component

Recommended folder structure (full):

```
global (Global resources such as IAM, SNS, S3)
   └ iam
   └ sns

stage (Non-production workloads, testing)
   └ vpc
   └ mysql
   └ frontend

prod (Production workloads, user-facing apps)
   └ vpc
   └ mysql
   └ frontend

mgmt (DevOps tooling such as Jenkins, Bastion
Host)
   └ vpc
```

Each component in each environment gets its own `.tfstate`

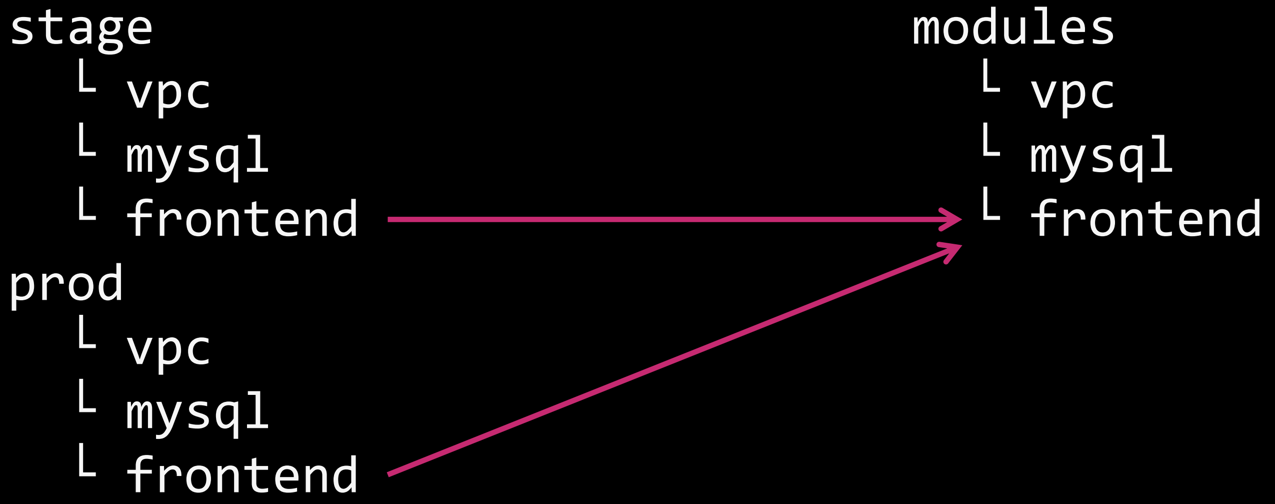
```
global (Global resources such as IAM, SNS, S3)
   └ iam
   └ sns

stage (Non-production workloads, testing)
   └ vpc
   └ mysql
   └ frontend

prod (Production workloads, user-facing apps)
   └ vpc
   └ mysql
   └ frontend

mgmt (DevOps tooling such as Jenkins, Bastion
Host)
   └ vpc
```

Use terraform_remote_state to share state between them

5. Use modules

```
stage
  └ vpc
  └ mysql
  └ frontend
prod
  └ vpc
  └ mysql
  └ frontend
```

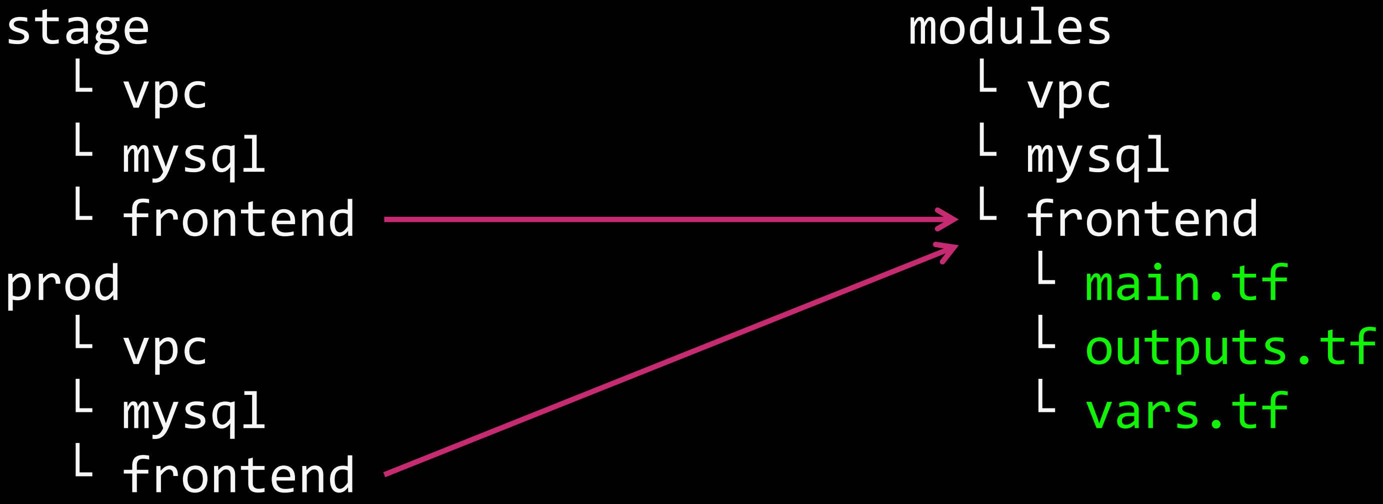
How do you avoid copy/pasting code between stage and prod?



```
stage
  └ vpc
  └ mysql
  └ frontend
prod
  └ vpc
  └ mysql
  └ frontend
```

```
modules
  └ vpc
  └ mysql
  └ frontend
```

Define reusable modules!



```
stage
  └ vpc
  └ mysql
  └ frontend
prod
  └ vpc
  └ mysql
  └ frontend
```

```
modules
  └ vpc
  └ mysql
  └ frontend
    └ main.tf
    └ outputs.tf
    └ vars.tf
```

Each module defines one reusable component

```
variable "name" {
    description = "The name of the EC2
instance"
}

variable "ami" {
    description = "The AMI to run on the EC2
instance"
}

variable "memory" {
    description = "The amount of memory to
allocate"
}
```

Define **inputs** in `vars.tf` to configure the module

```
module "frontend" {
    source = "./modules/frontend"

    name = "frontend-stage"
    ami = "ami-123asd1"
    memory = 512
}
```

Use the module in stage

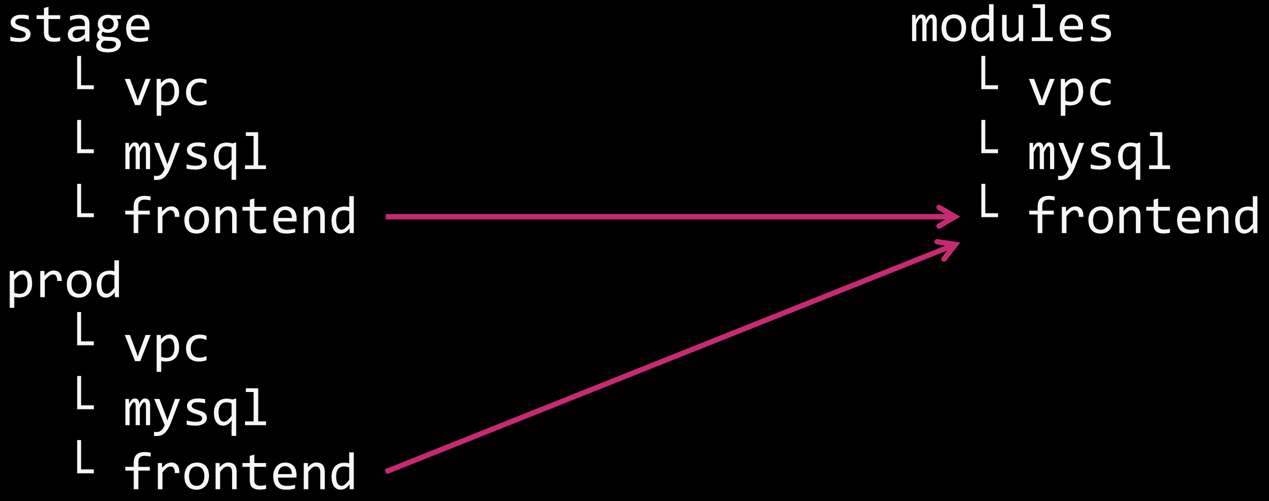
(stage/frontend/main.tf)

```
module "frontend" {
    source = "./modules/frontend"

    name = "frontend-prod"
    ami = "ami-123abcd"
    memory = 2048
}
```

And in prod

(prod/frontend/main.tf)

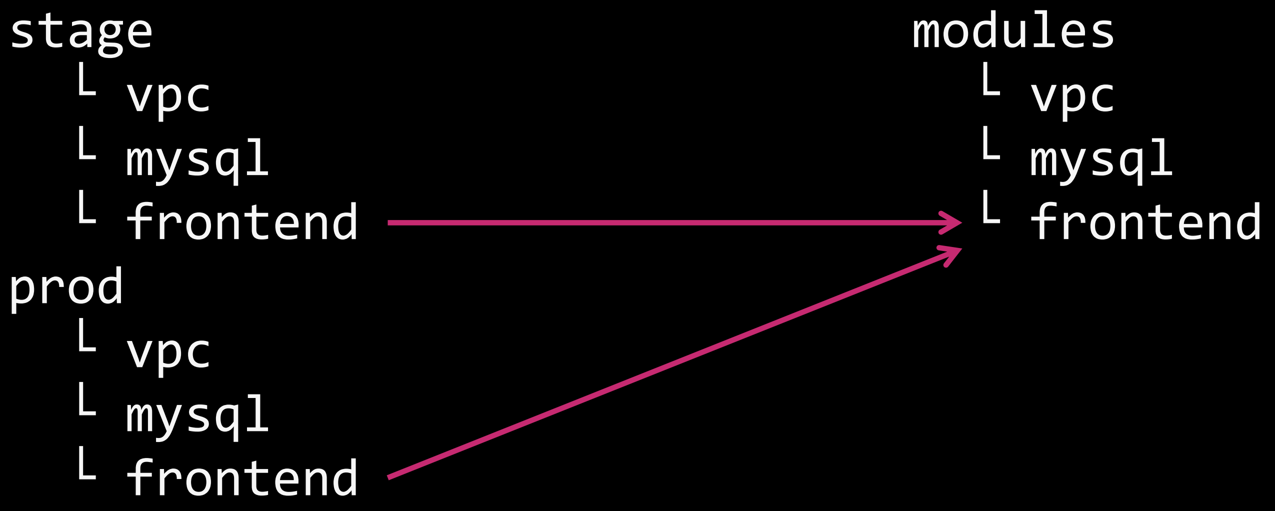
6. Use versioned modules



```
stage
   └ vpc
   └ mysql
   └ frontend
prod
   └ vpc
   └ mysql
   └ frontend
```

```
modules
   └ vpc
   └ mysql
   └ frontend
```

If stage and prod point to the same folder, you lose isolation



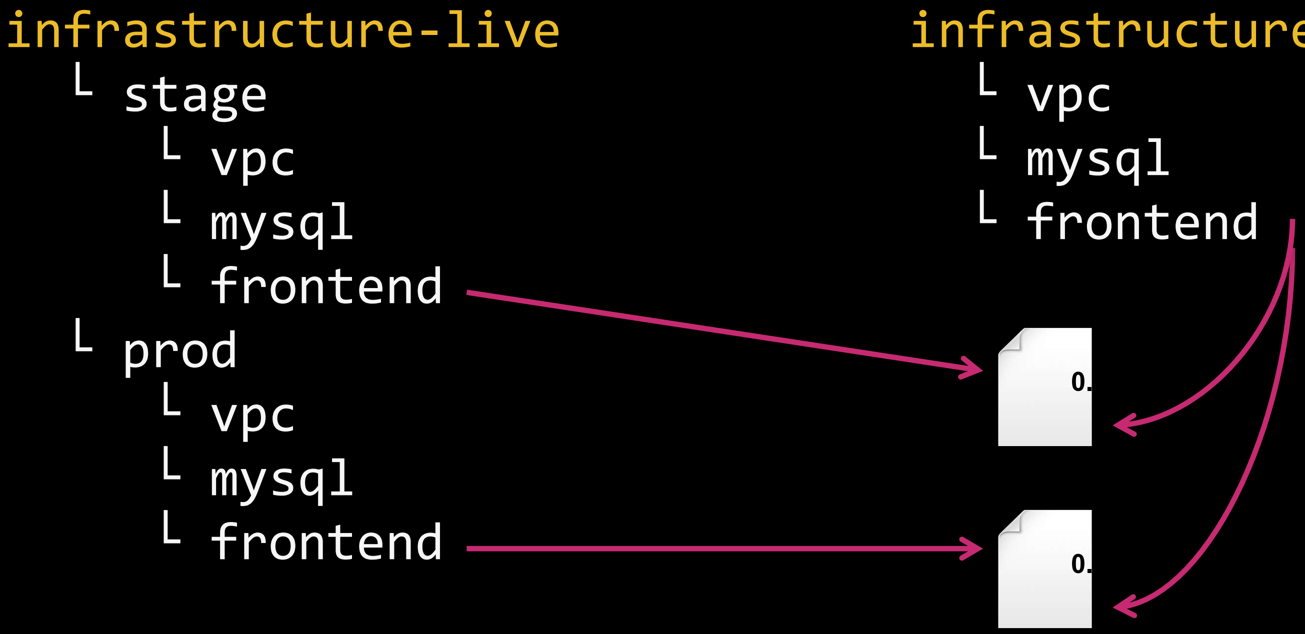
```
stage                          modules
  └ vpc                          └ vpc
  └ mysql                        └ mysql
  └ frontend                     └ frontend
prod
  └ vpc
  └ mysql
  └ frontend
```

Any change in modules/frontend **affects both stage and prod**

```
infrastructure-live
  └ stage
     └ vpc
     └ mysql
     └ frontend
  └ prod
     └ vpc
     └ mysql
     └ frontend
```

```
infrastructure-modules
  └ vpc
  └ mysql
  └ frontend
```

Solution: define modules in a separate repository



```
infrastructure-live
  └ stage
     └ vpc
     └ mysql
     └ frontend
  └ prod
     └ vpc
     └ mysql
     └ frontend
```

```
infrastructure-modules
  └ vpc
  └ mysql
  └ frontend
```

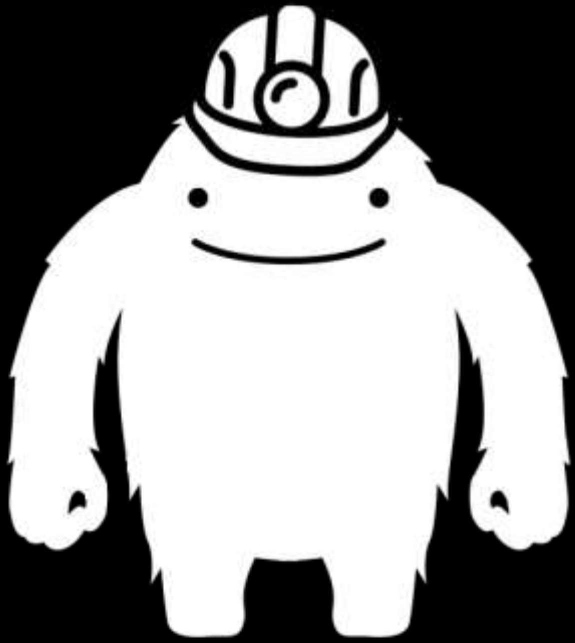
Now stage and prod can use different **versioned URLs**

```
module "frontend" {
    source = "git::git@github.com:foo/infrastructure-modules.git//frontend?ref=0.2"

    name = "frontend-prod"
    ami = "ami-123abcd"
    memory = 2048
}
```

Example Terraform code

(prod/frontend/main.tf)

7. State file storage



Use terragrunt

github.com/gruntwork-io/terragrunt

```
dynamoDbLock = {
  stateFileId = "mgmt/bastion-host"
}
```

Use a custom lock (`stateFileId`) for each set of templates

```
remoteState = {
  backend = "s3"
  backendConfigs = {
    bucket = "acme-co-terraform-state"
    key = "mgmt/bastion-host/terraform.tfstate"
    encrypt = "true"
  }
}
```

Use an S3 bucket with encryption for remote state storage

Terraform Loops

Terraform is Declarative. Very little logic is possible.

Count can be used limitedly

```
resource "aws_instance" "example" {
    count = 3
    ami = "${var.ami}"
    instance_type = "t2.micro"
    tags { Name = "${var.name}-${count.index}" }
}
```

Do even more with interpolation functions:

terraform.io/docs/configuration/interpolation.html

```
resource "aws_instance" "example" {
    count = 3
    ami = "${element(var.amis, count.index)}"
    instance_type = "t2.micro"
    tags { Name = "${var.name}-${count.index}" }
}

variable "amis" {
  type = "list"
  default = ["ami-abc123", "ami-abc456", "ami-
abc789"]
}
```

Create three EC2 Instances, each with a different AMI

```
output "all_instance_ids" {
    value = ["${aws_instance.example.*.id}"]
}

output "first_instance_id" {
    value = "${aws_instance.example.0.id}"
}
```

Note: resources with count are actually lists of resources!

Terraform if-else

Terraform is Declarative. Very little logic is possible.

Count can be used limitedly for same purpose

```
resource "aws_instance" "example" {
    count = "${var.should_create_instaance}"
    ami = "ami-abcd1234"
    instance_type = "t2.micro"
    tags { Name = "${var.name}" }
}

variable "should_create_instance" {   default = true }
```

Advantages of Terraform

1. Define infrastructure-as-code
2. Concise, readable syntax
3. Reuse: inputs, outputs, modules
4. Plan command!
5. Cloud agnostic
6. Very active development

Disadvantages of Terraform

1. **Maturity. You will hit bugs.**
2. **Collaboration on Terraform state is tricky (but not with terragrunt)**
3. **No rollback**
4. **Poor secrets management**

Thank You