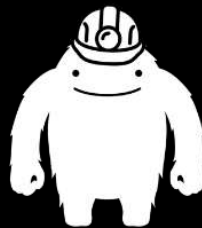


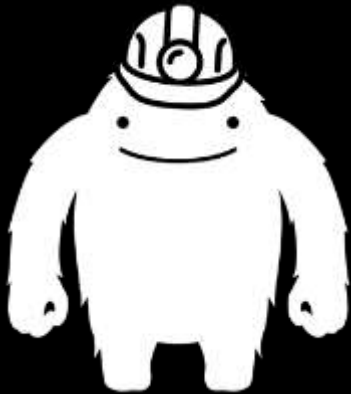


TERRAFORM COMPREHENSIVE TRAINING

Taught by
Gruntwork



<http://gruntwork.io>



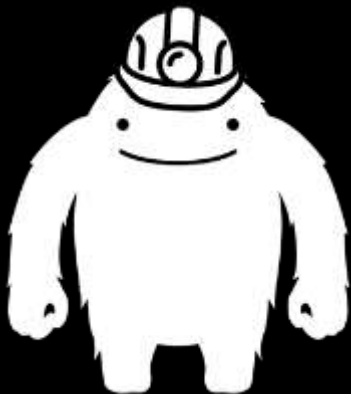
Gruntwork

<http://gruntwork.io>

We've pre-written Terraform packages for the most common AWS components.

We test, update, and support these packages.

When a software team purchases a package, they get 100% of the source code.



Gruntwork

<http://gruntwork.io>

Sample Packages

- Network Topology (VPC)
- Monitoring and Alerting
- Docker Cluster
- Continuous Delivery

Code samples:

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



Outline

1. **Intro**
2. **State**
3. **Modules**
4. **Best practices**
5. **Gotchas**
6. **Recap**

Outline

1. **Intro**
2. State
3. Modules
4. Best practices
5. Gotchas
6. Recap



TERRAFORM

**Terraform is a tool for
provisioning infrastructure**



PROVIDERS

Atlas
AWS
Azure (Service Management)
Azure (Resource Manager)
Chef
CenturyLinkCloud
CloudFlare
CloudStack
Consul
Datadog
DigitalOcean
DNSMadeEasy
DNSimple
Docker
Dyn
Github
Google Cloud

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.

It supports many **providers** (cloud agnostic)

[DOCUMENTATION HOME](#)[AWS PROVIDER](#)[EC2 RESOURCES](#)

[aws_ami](#)
[aws_ami_copy](#)
[aws_ami_from_instance](#)
[aws_app_cookie_stickiness_policy](#)
[aws_autoscaling_group](#)
[aws_autoscaling_lifecycle_hook](#)
[aws_autoscaling_notification](#)
[aws_autoscaling_policy](#)
[aws_autoscaling_schedule](#)
[aws_ebs_volume](#)
[aws_eip](#)
[aws_elb](#)
[aws_instance](#)

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:

- **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

And many **resources** for each
provider

You define resources as **code** in
Terraform templates



```
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



EC2 Management Console

https://console.aws.amazon.com/ec2/v2/home?region=us-east-1#Instances:search=terraform-example;sort=instancetype

AWS Services Edit

N. Virginia Support

EC2 Dashboard
Events
Tags
Reports
Limits

INSTANCES

Instances
Spot Requests
Reserved Instances
Scheduled Instances
Commands
Dedicated Hosts

IMAGES

AMIs
Bundle Tasks

ELASTIC BLOCK STORE

Volumes
Snapshots

Launch Instance Connect Actions

search : terraform-example Add filter

Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS
terraform-example	i-f3d58c70	t2.micro	us-east-1d	running	Initializing	None	ec2-54-88-184-

Instance: i-f3d58c70 (terraform-example) Public DNS: ec2-54-88-184-154.compute-1.amazonaws.com

Feedback English

© 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

Now the EC2 instance is running!



You can parameterize your
templates using **variables**



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

**Define a variable. description,
default, and type are optional.**




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

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

Note the use of `${}` syntax to reference `var.name` in tags



```
> terraform plan
```

```
var.name
```

```
Enter a value: foo
```

```
~ aws_instance.example
```

```
tags.Name: "terraform-example" => "foo"
```

Use **plan** to verify your changes. It prompts you for the variable.



```
> terraform apply -var name=foo
aws_instance.example: Refreshing state...
aws_instance.example: Modifying...
tags.Name: "terraform-example" => "foo"
aws_instance.example: Modifications complete

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

**You can also pass variables using
the `-var` parameter**



**You can create dependencies
between resources**

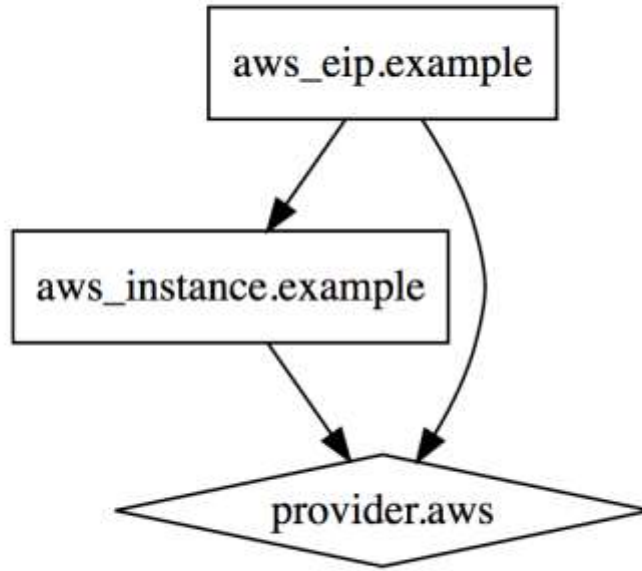


```
resource "aws_eip" "example" {  
    instance = "${aws_instance.example.id}"  
}
```

```
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**



You can **clean up all resources
you created with Terraform**



```
> 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.

Just use the **destroy** command




```
> 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.
```

But how did Terraform know what to destroy?



Outline

1. Intro
- 2. State**
3. Modules
4. Best practices
5. Gotchas
6. Recap

Terraform records **state of
everything it has done**



```
> ls -al
```

```
-rw-r--r-- 6024 Apr 5 17:58 terraform.tfstate
```

```
-rw-r--r-- 6024 Apr 5 17:58 terraform.tfstate.backup
```

**By default, state is stored locally
in `.tfstate` files**



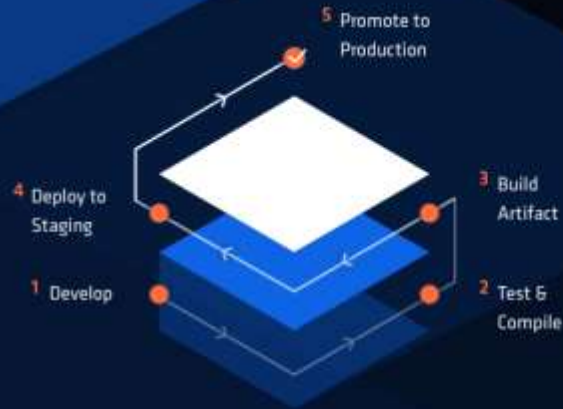
```
> terraform remote config \  
-backend=s3 \  
-backend-config=bucket=my-s3-bucket \  
-backend-config=key=terraform.tfstate \  
-backend-config=encrypt=true \  
-backend-config=region=us-east-1
```

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



Atlas is the HashiCorp Suite for the Enterprise.

Enable developers to rapidly deploy with an automated, policy-enforced workflow. Operators use Atlas's flexibility to tailor the platform to their environment, while still presenting a simple self-service experience to developers.

[REQUEST A DEMO >](#)

Only Atlas provides **locking**, but it can be expensive



Jenkins

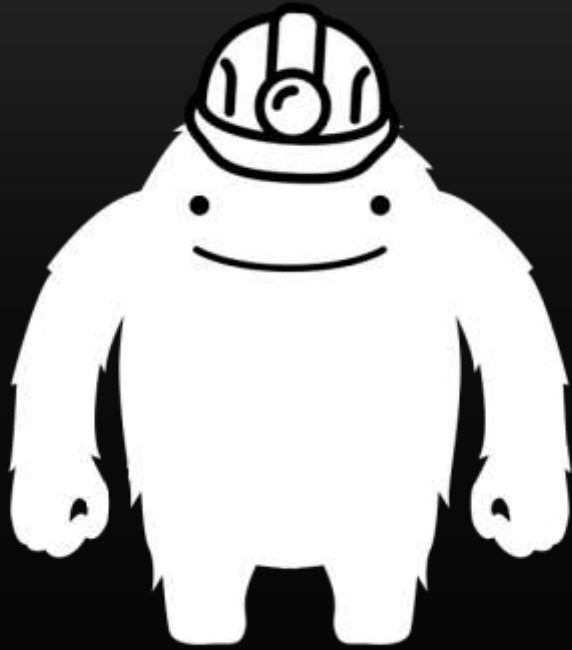
Build great things at any scale

The leading open source automation server, Jenkins provides hundreds of plugins to support building, deploying and automating any project.

[Download Jenkins](#)

Get 1.642.4 LTS .war or the latest 1.656 weekly release

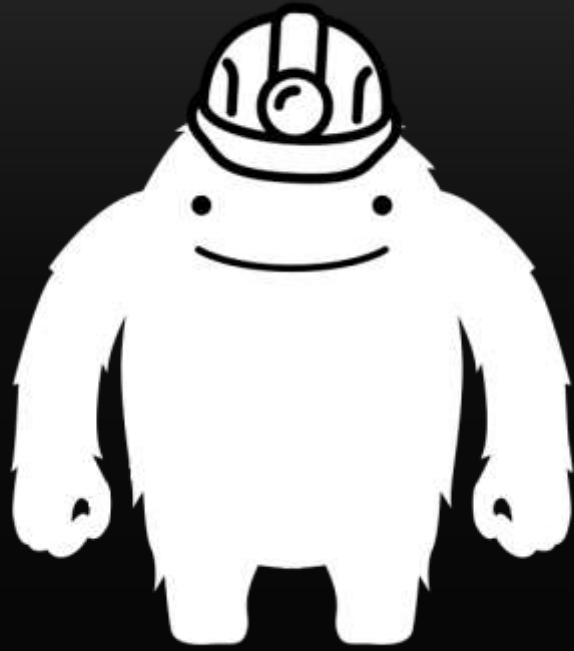
One alternative: **manual coordination** (+ CI job)



Better alternative: **terragrunt**

github.com/gruntwork-io/terragrunt





Terragrunt is a thin, open source
wrapper for Terraform





It provides locking using
DynamoDB



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

Terragrunt looks for a .terragrunt file for its configuration.

- > terragrunt plan
- > terragrunt apply
- > terragrunt destroy

**Use all the normal Terraform
commands with Terragrunt**

```
> terragrunt apply
[terragrunt] Acquiring lock for bastion-host in DynamoDB
[terragrunt] Running command: terraform apply

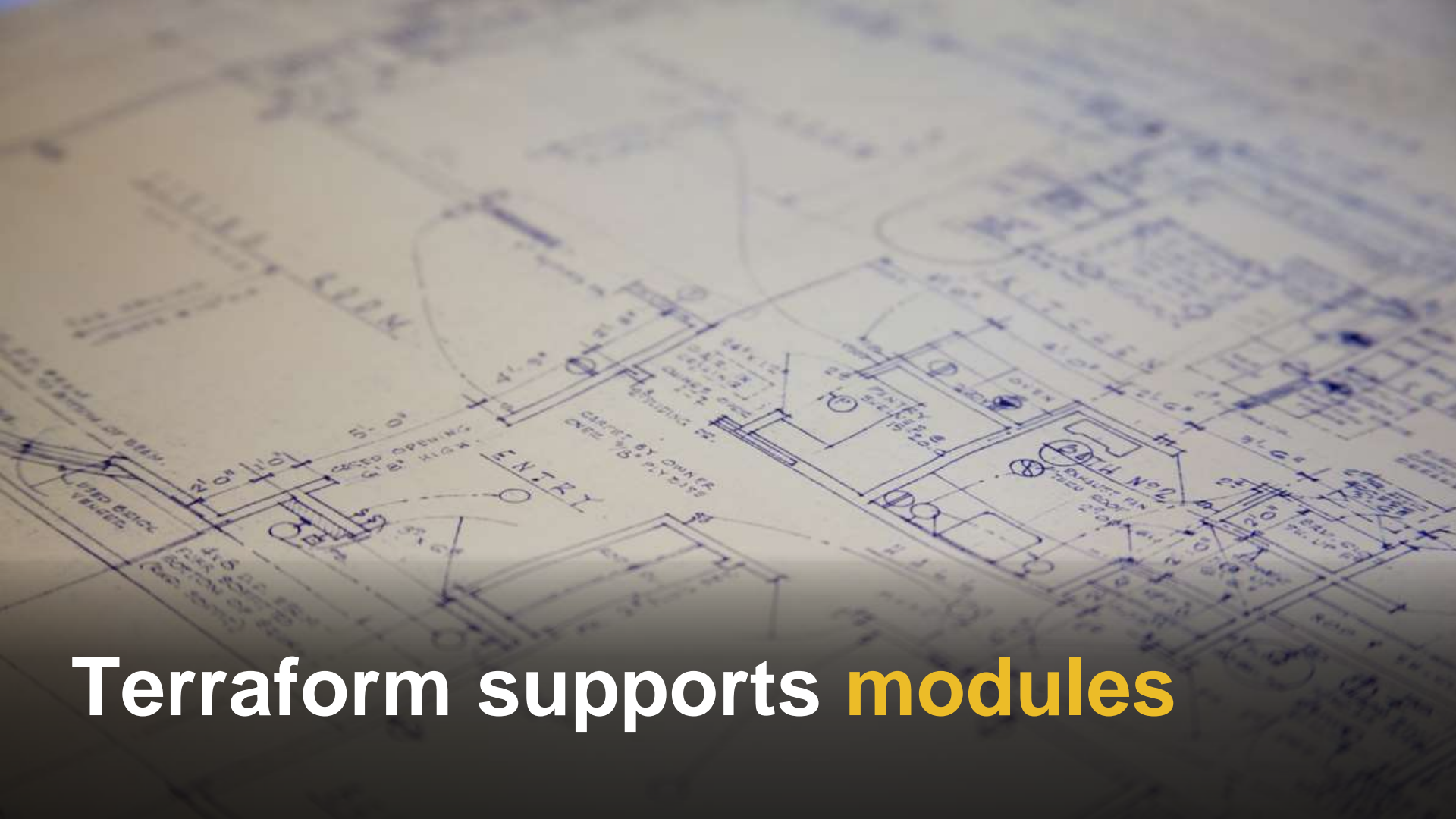
aws_instance.example: Creating...
  ami:                "" => "ami-0d729a60"
  instance_type:      "" => "t2.micro"
[...]
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

```
[terragrunt] Releasing lock for bastion-host in DynamoDB
```

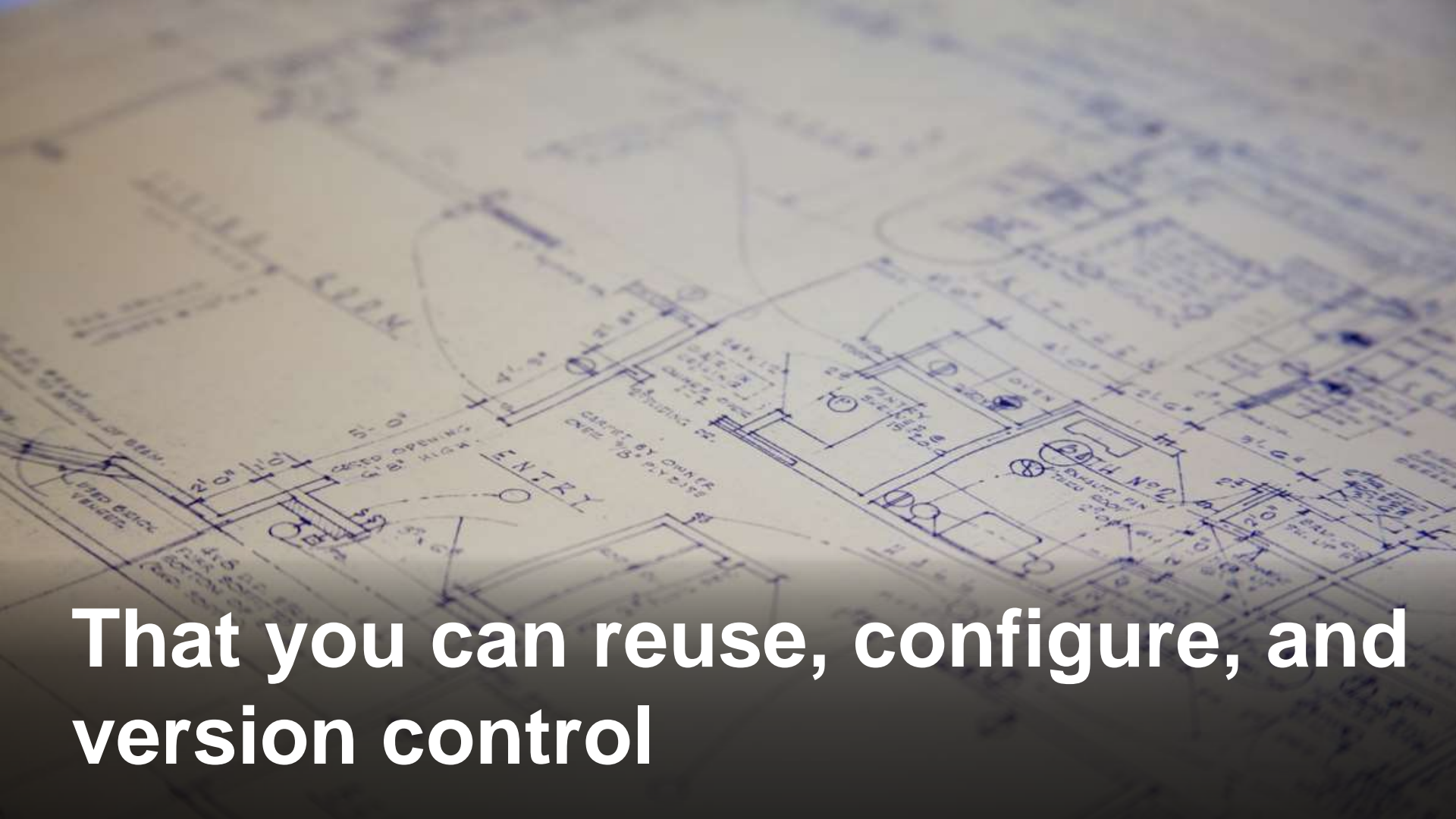
Terragrunt automatically acquires and releases locks on apply/destroy

Outline

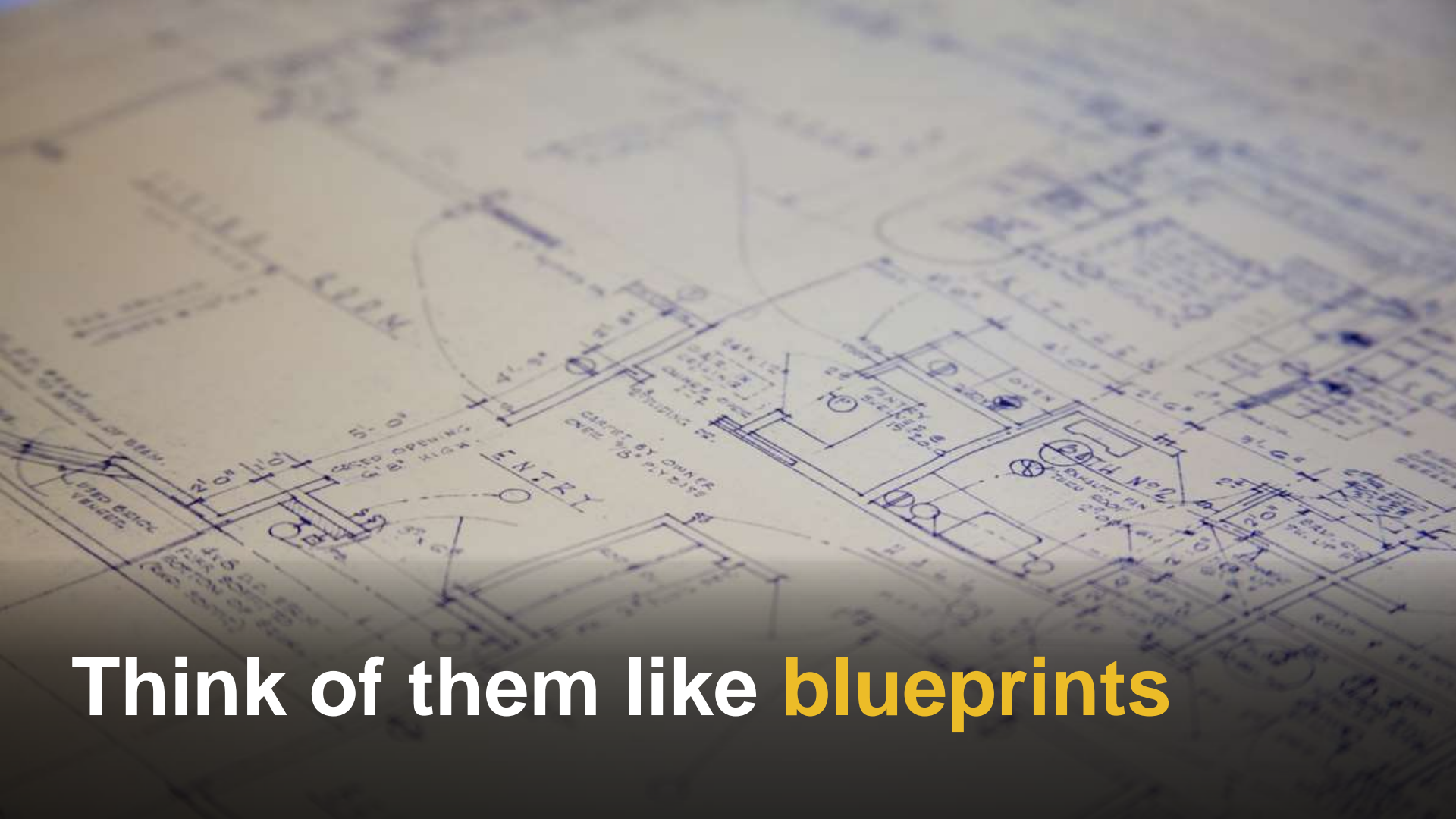
1. Intro
2. State
3. **Modules**
4. Best practices
5. Gotchas
6. Recap



Terraform supports **modules**



**That you can reuse, configure, and
version control**



Think of them like **blueprints**

**A module is just a folder with
Terraform templates**



**Most Gruntwork Infrastructure
Packages are Terraform modules**



Our conventions:



1. vars.tf
2. main.tf
3. outputs.tf



```
variable "name" {  
    description = "The name of the EC2 instance"  
}  
  
variable "ami" {  
    description = "The AMI to run on the EC2 instance"  
}  
  
variable "port" {  
    description = "The port to listen on for HTTP requests"  
}
```

Specify module `inputs` in `vars.tf`



```
resource "aws_instance" "example" {  
    ami = "${var.ami}"  
    instance_type = "t2.micro"  
    user_data = "${template_file.user_data.rendered}"  
    tags { Name = "${var.name}" }  
}
```

Create **resources** in **main.tf**



```
output "url" {  
    value = "http://${aws_instance.example.ip}:${var.port}"  
}
```

Specify outputs in outputs.tf



See example modules:

gruntwork.io/#what-we-do



Using a module:



```
module "example_rails_app" {  
  source = "../rails-module"  
}
```

The **source** parameter specifies
what module to use



```
module "example_rails_app" {  
  source = "git::git@github.com:foo/bar.git//module?ref=0.1"  
}
```

It can even point to a **versioned**
Git URL



```
module "example_rails_app" {  
    source = "git::git@github.com:foo/bar.git//module?ref=0.1"  
  
    name = "Example Rails App"  
    ami = "ami-123asd1"  
    port = 8080  
}
```

**Specify the module's inputs like
any other Terraform resource**



```
module "example_rails_app_stg" {  
  source = "../rails-module"  
  name = "Example Rails App staging"  
}  
  
module "example_rails_app_prod" {  
  source = "../rails-module"  
  name = "Example Rails App production"  
}
```

You can reuse the same module multiple times



```
> terraform get -update
```

```
Get: file:///home/ubuntu/modules/rails-module
```

```
Get: file:///home/ubuntu/modules/rails-module
```

```
Get: file:///home/ubuntu/modules/asg-module
```

```
Get: file:///home/ubuntu/modules/vpc-module
```

**Run the `get` command before
running `plan` or `apply`**



Outline

1. Intro
2. State
3. Modules
4. **Best practices**
5. Gotchas
6. Recap

1. Plan **before** apply



2. Stage before prod



3. Isolated environments





**It's tempting to define everything
in 1 template**





But then a mistake anywhere
could break **everything**





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

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)

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)

Use terraform_remote_state to share state between them

4. Isolated components





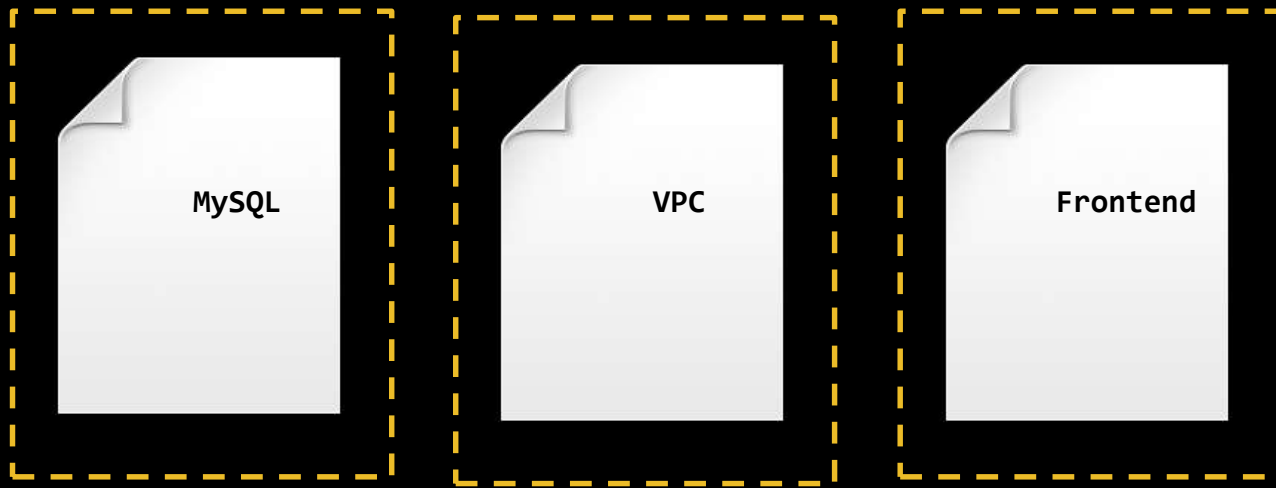
**It's tempting to define everything
in 1 template**





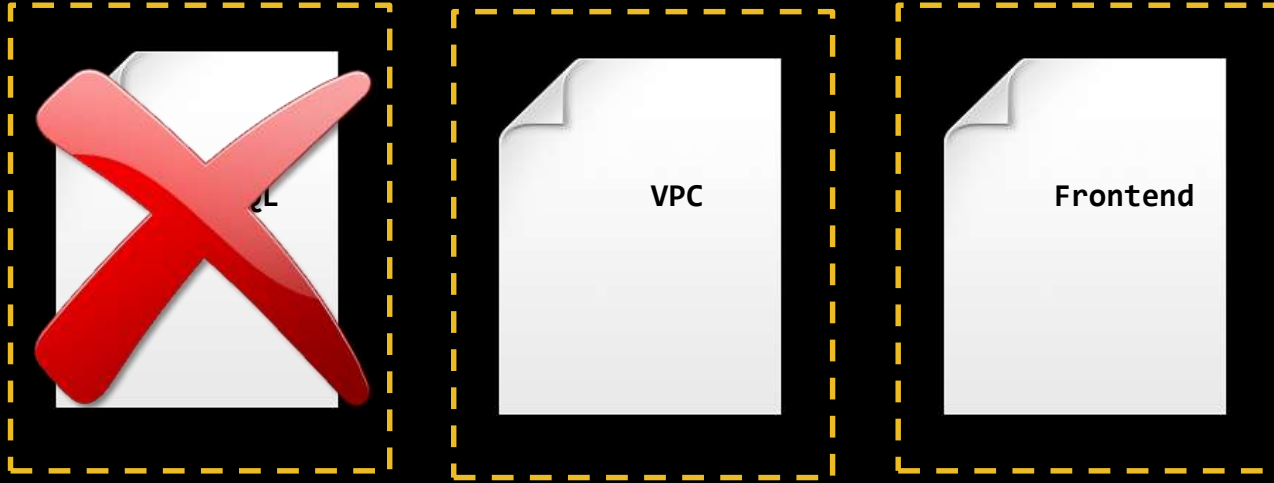
But then a mistake anywhere
could break **everything**





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





**That way, a problem in MySQL
doesn't affect the whole VPC**



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

- └ bastion

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

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

mgmt (DevOps tooling such as Jenkins, Bastion Host)

- └ bastion

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

Use terraform_remote_state to share state between them

mgmt (DevOps tooling such as Jenkins, Bastion Host)

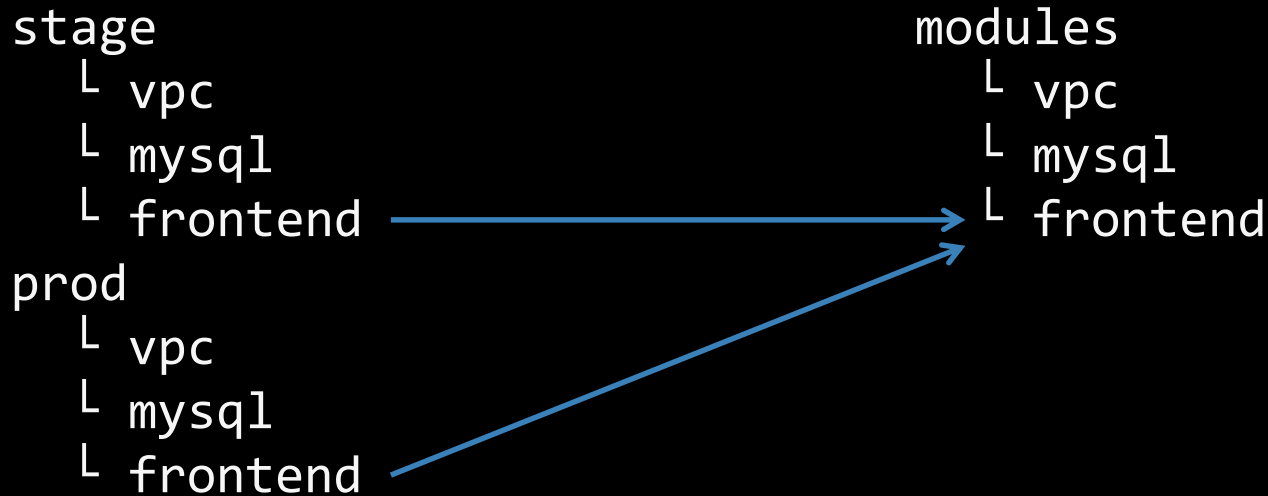
- └ bastion

5. Use modules

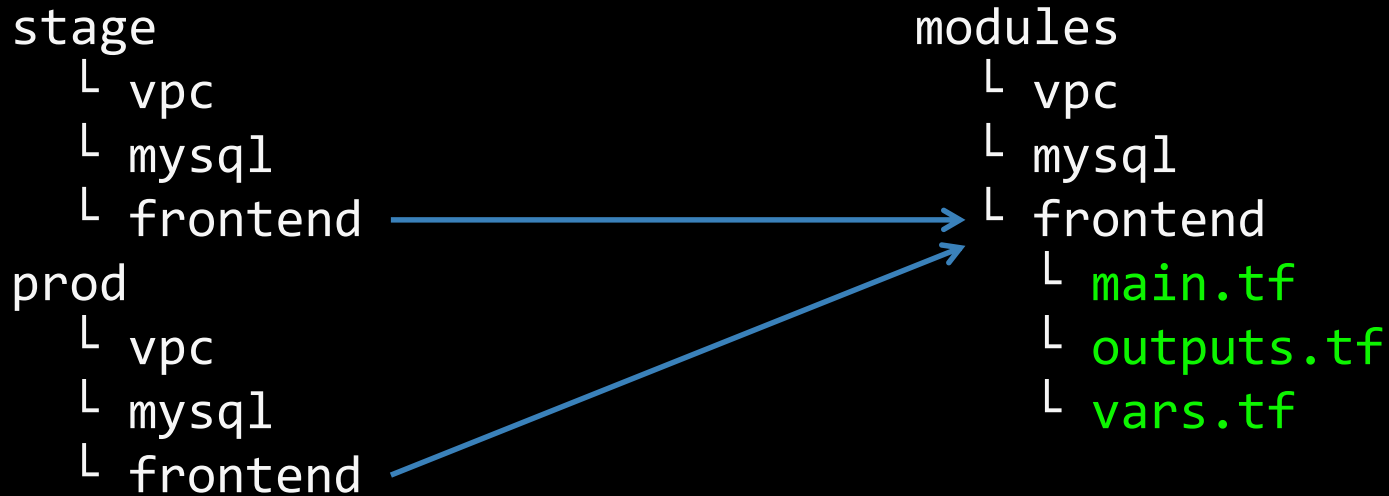


```
stage
├─ vpc
├─ mysql
├─ frontend
prod
├─ vpc
├─ mysql
├─ frontend
```

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



Define reusable modules!



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  
}
```

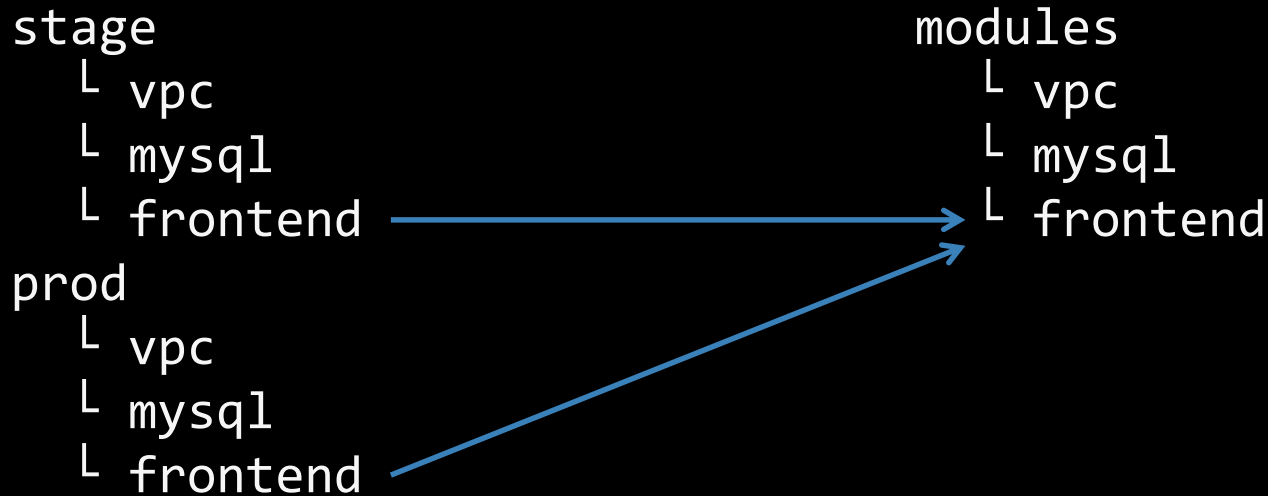
And in prod

(prod/frontend/main.tf)

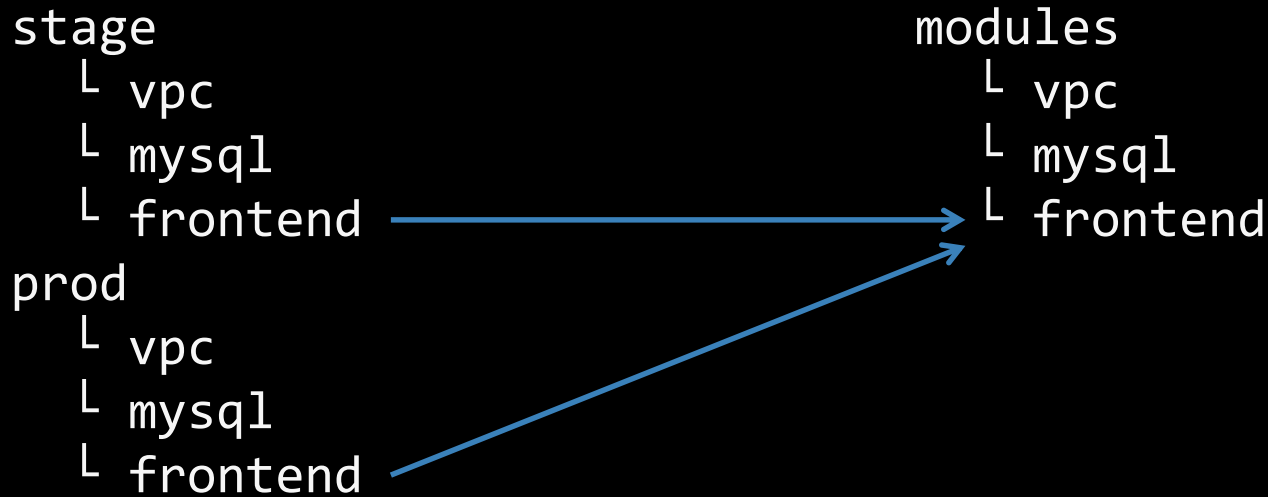


6. Use versioned modules





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



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**

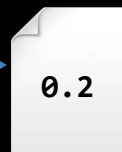
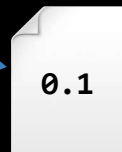
infrastructure-live

- └ stage
 - └ vpc
 - └ mysql
 - └ frontend

- └ prod
 - └ vpc
 - └ mysql
 - └ frontend

infrastructure-modules

- └ vpc
- └ mysql
- └ frontend



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

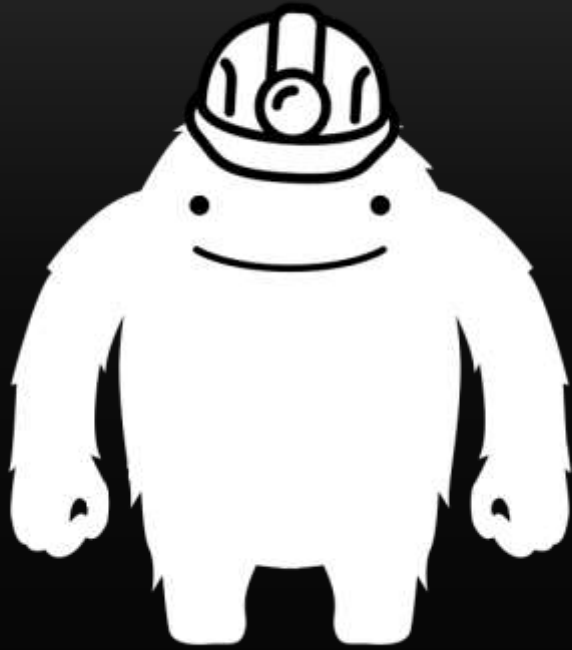

```
module "frontend" {  
    source =  
    "git::git@github.com:foo/infrastructure-  
modules.git//frontend?ref=0.2"  
  
    name = "frontend-prod"  
    ami = "ami-123abcd"  
    memory = 2048  
}
```

Example Terraform code
(prod/frontend/main.tf)



7. State file storage





Use **terragrunt**

github.com/gruntwork-io/terragrunt

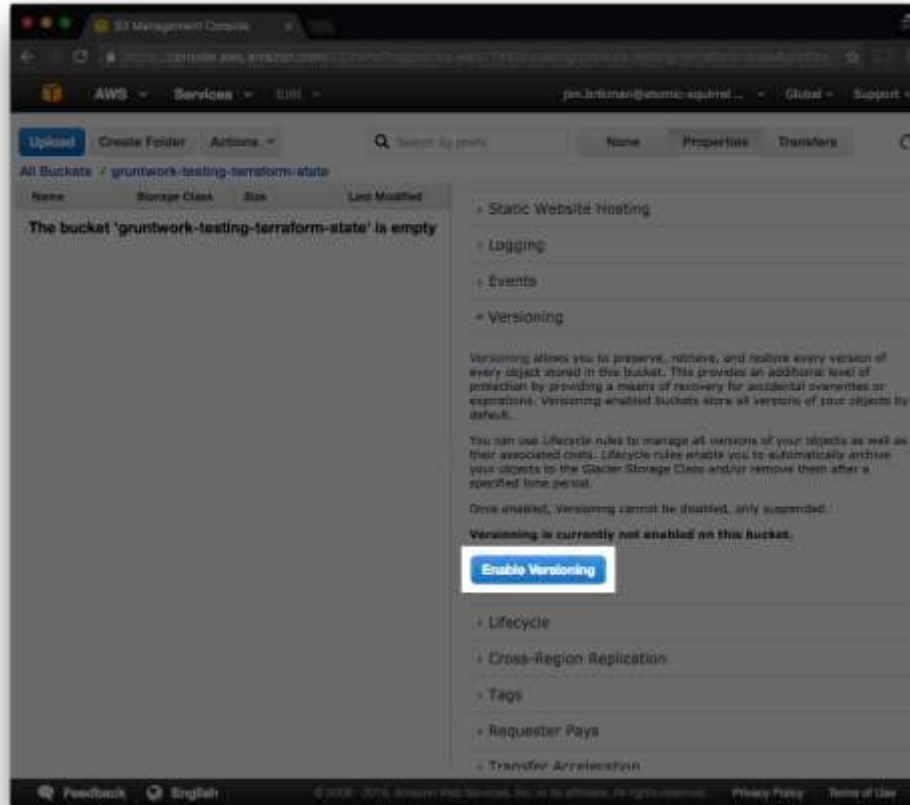


```
dynamoDbLock = {  
    stateFileId = "mgmt/bastion-host"  
}
```

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



Enable versioning on the S3 bucket!



7. Loops



Terraform is **declarative**, so very little “logic” is possible...



But you can “loop” to create
multiple resources using **count**



```
resource "aws_instance" "example" {  
    count = 1  
    ami = "${var.ami}"  
    instance_type = "t2.micro"  
    tags { Name = "${var.name}" }  
}
```

Create **one** EC2 Instance



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

Create **three** EC2 Instances



Use `count.index` to modify each
“iteration”



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

Create three EC2 Instances, each
with a **different name**



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!**



8. If-statements



Terraform is **declarative**, so very little “logic” is possible...



But you can do a limited form of
if-statement using count



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

Note the use of a `boolean` in the `count` parameter



In **HCL**:

- `true` = 1
- `false` = 0



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

**So this creates 1 EC2 Instance if
`should_create_instance` is true**



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

**Or 0 EC2 Instances if
should_create_instance is false**



It's equivalent to:

```
if (should_create_instance)  
    create_instance()
```



**There are many permutations of
this trick (e.g. using length)**



Outline

1. Intro
2. State
3. Modules
4. Best practices
5. Gotchas
6. Recap

1. Valid plans can fail

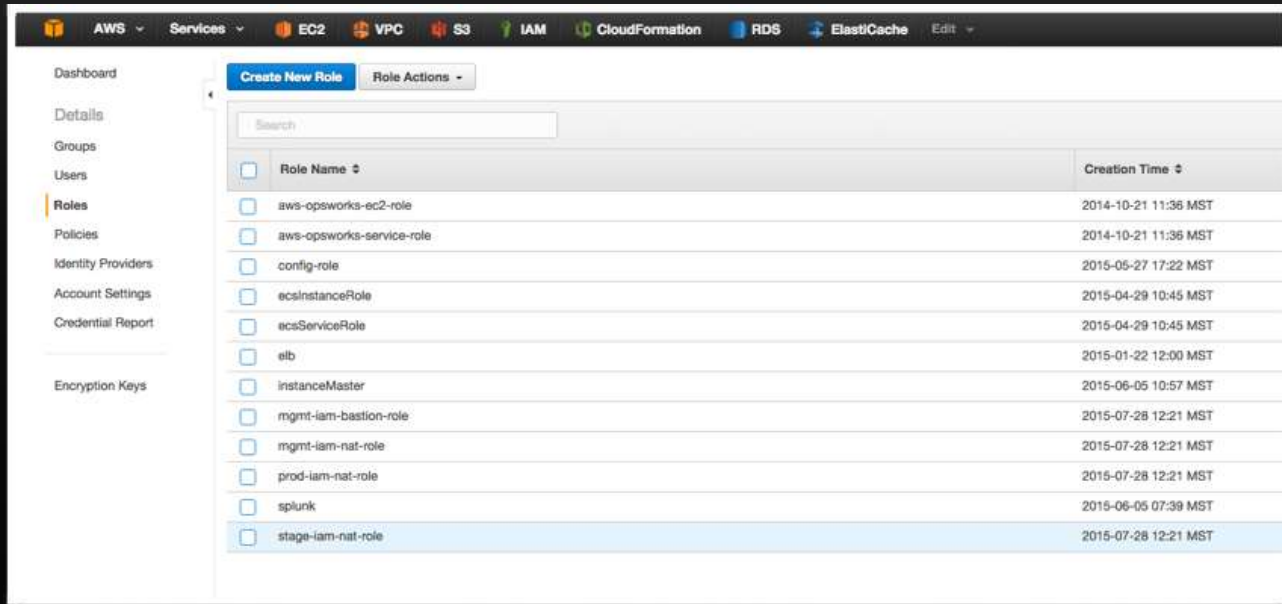


```
> terraform plan
+ aws_iam_instance_profile.instance_profile
  arn:                "<computed>"
  create_date:        "<computed>"
  name:               "stage-iam-nat-role"
  path:               "/"
  roles.2760019627:   "stage-iam-nat-role"
  unique_id:          "<computed>"
```

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

Valid plan to create IAM instance profiles





The screenshot shows the AWS IAM console interface. The top navigation bar includes links for AWS, Services, EC2, VPC, S3, IAM, CloudFormation, RDS, and ElastiCache. The left sidebar contains a menu with options: Dashboard, Details, Groups, Users, Roles (highlighted), Policies, Identity Providers, Account Settings, Credential Report, and Encryption Keys. The main content area is titled 'Create New Role' and 'Role Actions'. It features a search bar and a table of existing roles.

<input type="checkbox"/>	Role Name ↕	Creation Time ↕
<input type="checkbox"/>	aws-opsworks-ec2-role	2014-10-21 11:36 MST
<input type="checkbox"/>	aws-opsworks-service-role	2014-10-21 11:36 MST
<input type="checkbox"/>	config-role	2015-05-27 17:22 MST
<input type="checkbox"/>	ecsInstanceRole	2015-04-29 10:45 MST
<input type="checkbox"/>	ecsServiceRole	2015-04-29 10:45 MST
<input type="checkbox"/>	elb	2015-01-22 12:00 MST
<input type="checkbox"/>	instanceMaster	2015-06-05 10:57 MST
<input type="checkbox"/>	mgmt-iam-bastion-role	2015-07-28 12:21 MST
<input type="checkbox"/>	mgmt-iam-nat-role	2015-07-28 12:21 MST
<input type="checkbox"/>	prod-iam-nat-role	2015-07-28 12:21 MST
<input type="checkbox"/>	splunk	2015-06-05 07:39 MST
<input type="checkbox"/>	stage-iam-nat-role	2015-07-28 12:21 MST

But the instance profile already exists in IAM!



```
> terraform apply
```

Error applying plan:

```
* Error creating IAM role stage-iam-nat-role:  
EntityAlreadyExists: Role with name stage-iam-nat-role already  
exists  
    status code: 409, requestId: [e6812c4c-6fac-495c-be9d]
```

You get an error



Conclusion: Never make out-of-band changes.



2. AWS is eventually consistent



**Terraform doesn't always wait for
a resource to propagate**



**Which causes a variety of
intermittent bugs:**



```
> terraform apply
```

```
...
```

```
* aws_route.internet-gateway:  
error finding matching route for Route table (rtb-5ca64f3b)  
and destination CIDR block (0.0.0.0/0)
```



```
> terraform apply
```

```
...
```

```
* Resource 'aws_eip.nat' does not have attribute 'id' for  
variable 'aws_eip.nat.id'
```



```
> terraform apply
```

```
...
```

```
* aws_subnet.private-persistence.2: InvalidSubnetID.NotFound:  
The subnet ID 'subnet-xxxxxxx' does not exist
```



```
> terraform apply
```

```
...
```

```
* aws_route_table.private-persistence.2:
```

```
InvalidRouteTableID.NotFound: The routeTable ID 'rtb-2d0d2f4a'  
does not exist
```



```
> terraform apply
```

```
...
```

```
* aws_iam_instance_profile.instance_profile: diffs didn't  
match during apply. This is a bug with Terraform and should be  
reported.
```

```
* aws_security_group.asg_security_group_stg: diffs didn't  
match during apply. This is a bug with Terraform and should be  
reported.
```

**The most generic one: diffs didn't
match during apply**



**Most of these are harmless. Just
re-run `terraform apply`.**



**And try to run Terraform close to
your AWS region (replica lag)**



3. Avoid inline resources



```
resource "aws_route_table" "main" {  
  vpc_id = "${aws_vpc.main.id}"  
  
  route {  
    cidr_block = "10.0.1.0/24"  
    gateway_id = "${aws_internet_gateway.main.id}"  
  }  
}
```

**Some resources allow blocks to
be defined **inline**...**



```
resource "aws_route_table" "main" {  
  vpc_id = "${aws_vpc.main.id}"  
}  
  
resource "aws_route" "internet" {  
  route_table_id = "${aws_route_table.main.id}"  
  cidr_block = "10.0.1.0/24"  
  gateway_id = "${aws_internet_gateway.main.id}"  
}
```

Or in a **separate resource**



```
resource "aws_route_table" "main" {  
  vpc_id = "${aws_vpc.main.id}"  
}  
  
resource "aws_route" "internet" {  
  route_table_id = "${aws_route_table.main.id}"  
  cidr_block = "10.0.1.0/24"  
  gateway_id = "${aws_internet_gateway.main.id}"  
}
```

**Pick one technique or the other
(separate resource is preferable)**



```
resource "aws_route_table" "main" {  
  vpc_id = "${aws_vpc.main.id}"  
}  
  
resource "aws_route" "internet" {  
  route_table_id = "${aws_route_table.main.id}"  
  cidr_block = "10.0.1.0/24"  
  gateway_id = "${aws_internet_gateway.main.id}"  
}
```

**If you use both, you'll get
confusing errors!**



Affected resources:

- **aws_route_table**
- **aws_security_group**
- **aws_elb**
- **aws_network_acl**



4. Count interpolation



There is a significant **limitation**
on the count parameter:



**You cannot compute count from
dynamic data**



```
data "aws_availability_zones" "zones" {}

resource "aws_subnet" "public" {
  count = "${length(data.aws_availability_zones.zones.names)}"
  cidr_block = "${cidrsubnet(var.cidr_block, 5, count.index)}"
  availability_zone =
    "${element(data.aws_availability_zones.zones.names,
               count.index)}"
}
```

Example: this code won't work



```
> terraform apply
```

```
...
```

```
* strconv.ParseInt: parsing
```

```
"${length(data.aws_availability_zones.zones.names)}": invalid  
syntax
```



```
resource "aws_subnet" "public" {  
  count = "${length(data.aws_availability_zones.zones.names)}"  
  cidr_block = "${cidrsubnet(var.cidr_block, 5, count.index)}"  
  availability_zone =  
    "${element(data.aws_availability_zones.zones.names,  
               count.index)}"  
}
```

**data.aws_availability_zones
won't work since it fetches data**



```
resource "aws_subnet" "public" {  
  count = 3  
  cidr_block = "${cidrsubnet(var.cidr_block, 5, count.index)}"  
  availability_zone =  
    "${element(data.aws_availability_zones.zones.names,  
               count.index)}"  
}
```

A fixed number is OK



```
resource "aws_subnet" "public" {  
  count = "${var.num_availability_zones}"  
  cidr_block = "${cidrsubnet(var.cidr_block, 5, count.index)}"  
  availability_zone =  
    "${element(data.aws_availability_zones.zones.names,  
               count.index)}"  
}  
  
variable "num_availability_zones" {  
  default = 3  
}
```

So is a hard-coded variable



For more info, see:

github.com/hashicorp/terraform/issues/3888



Outline

1. Intro
2. State
3. Modules
4. Best practices
5. Gotchas
- 6. Recap**

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**



Questions?

Want to know when we share additional DevOps training?
Sign up for our Newsletter.

gruntwork.io/newsletter

