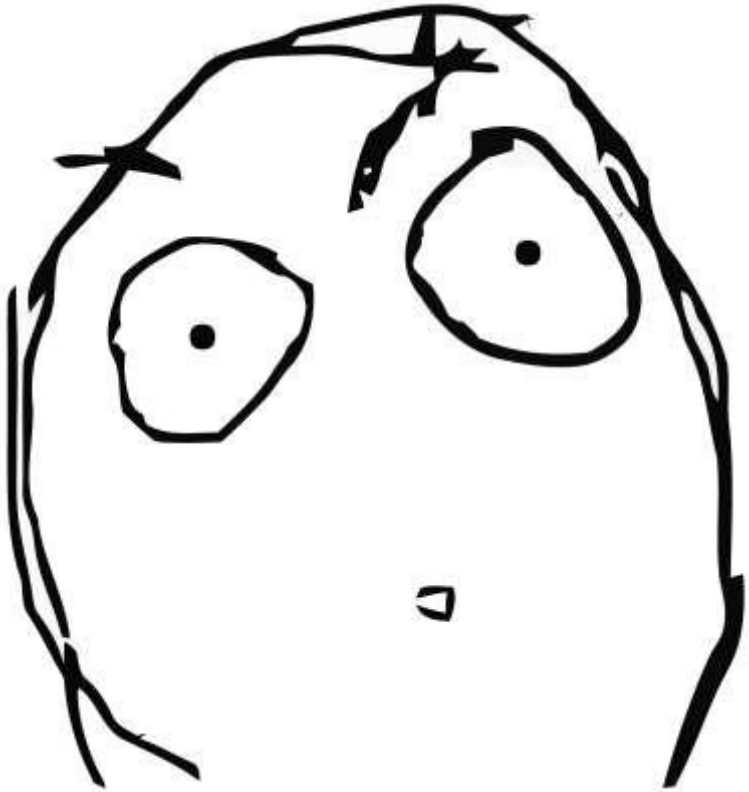
*Reusable, composable, battle-tested*

# TERRAFORM MODULES

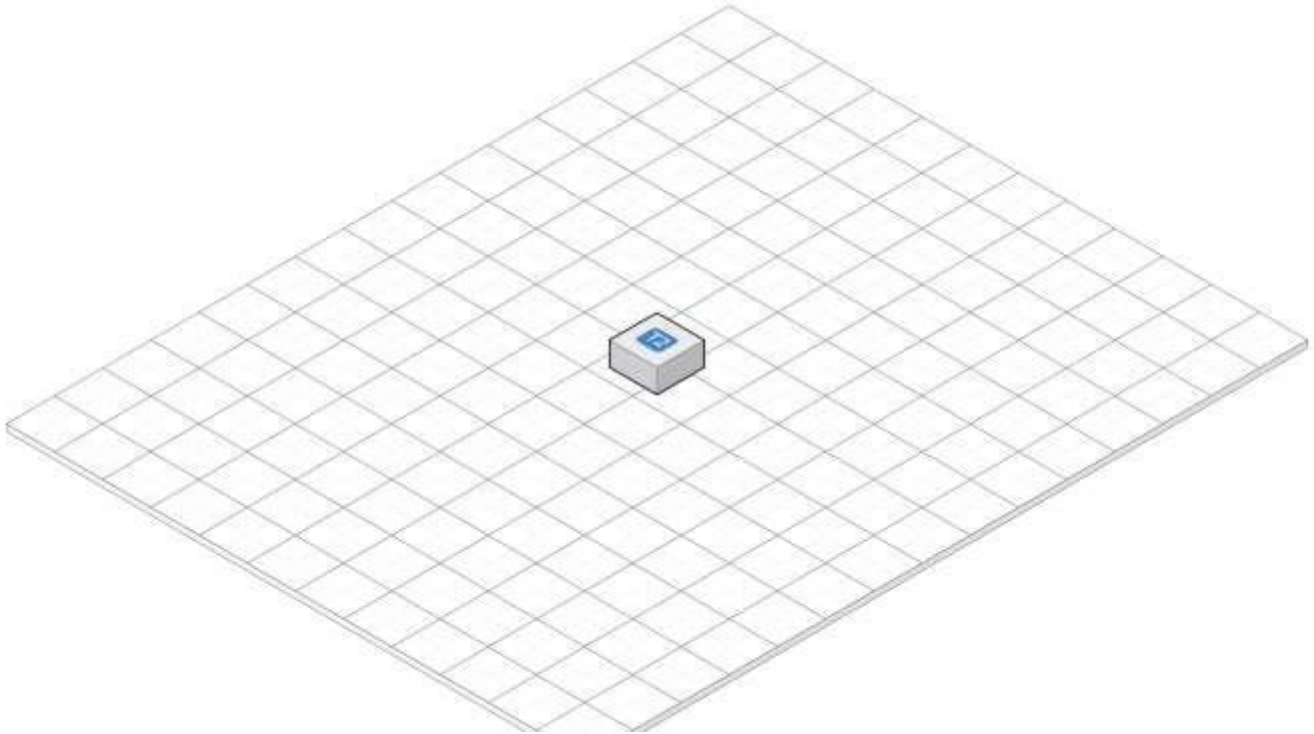
# Your project is code complete and it’s time to deploy it!

# I know, I'll use AWS!

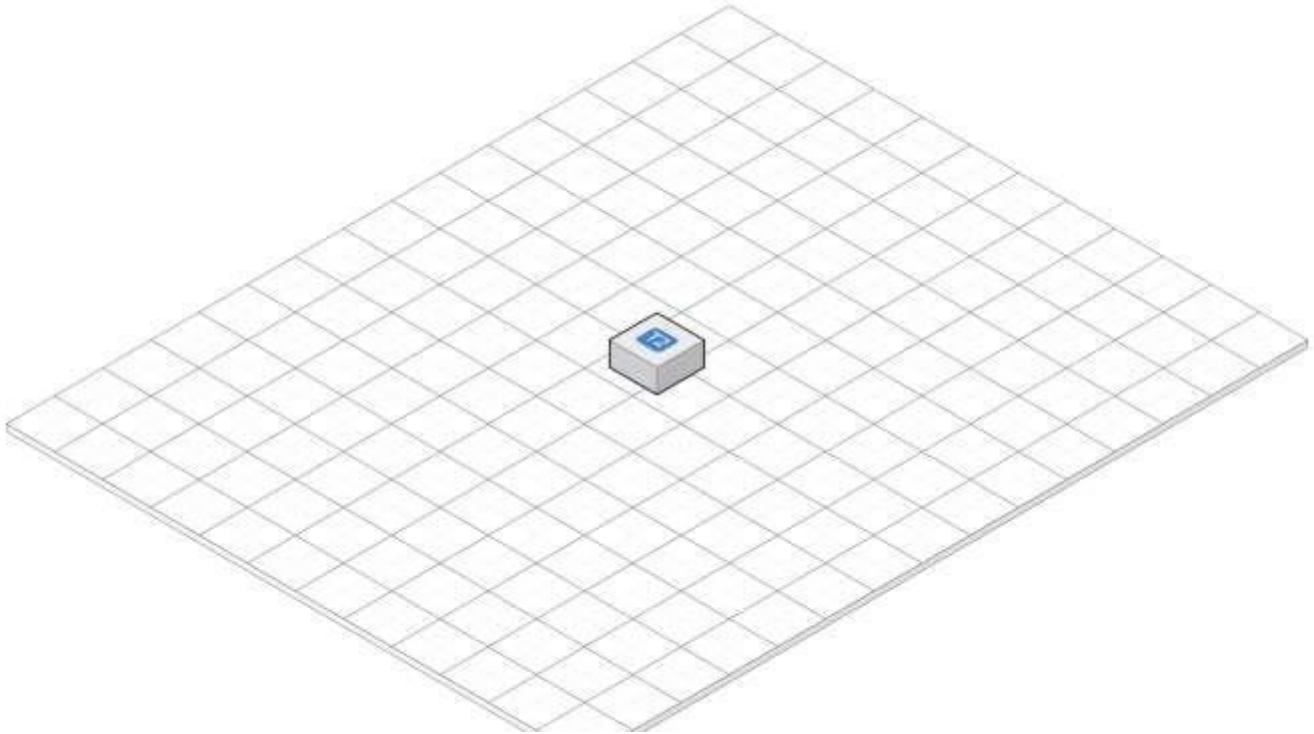

# You login to the AWS console



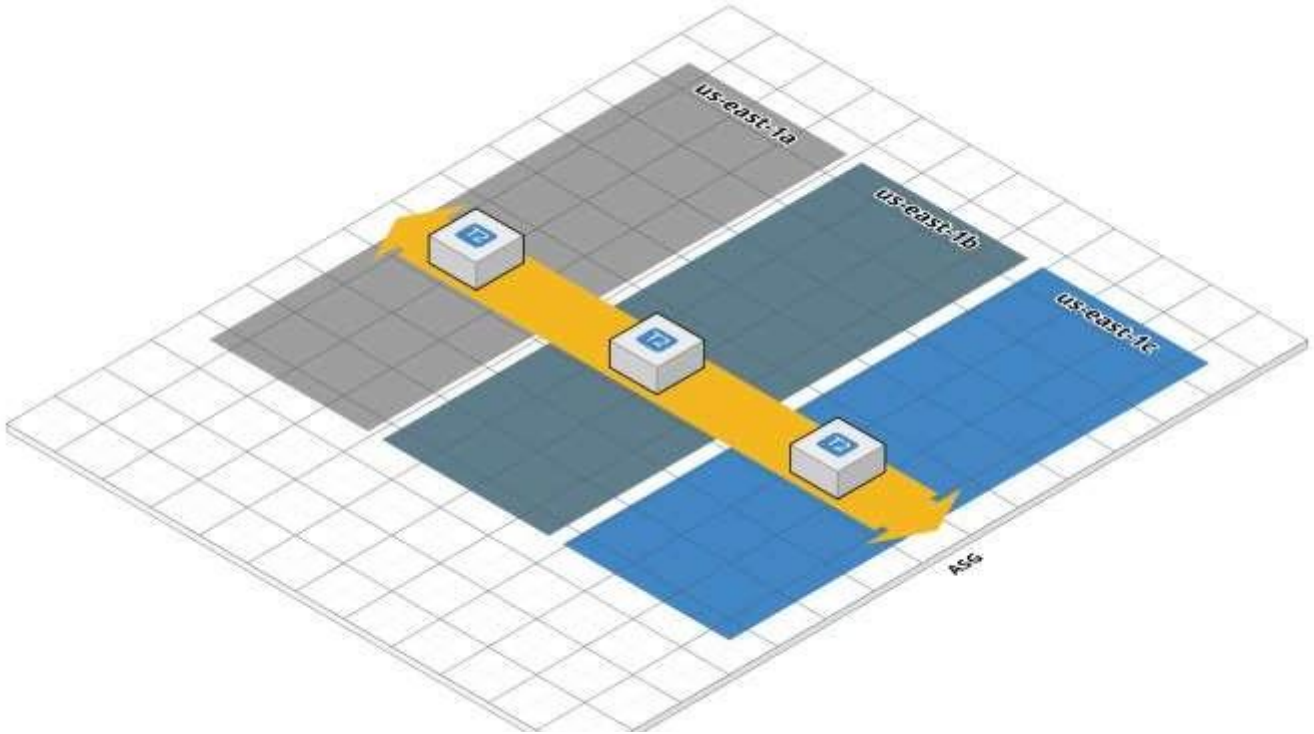
# (Spend hours reading docs)



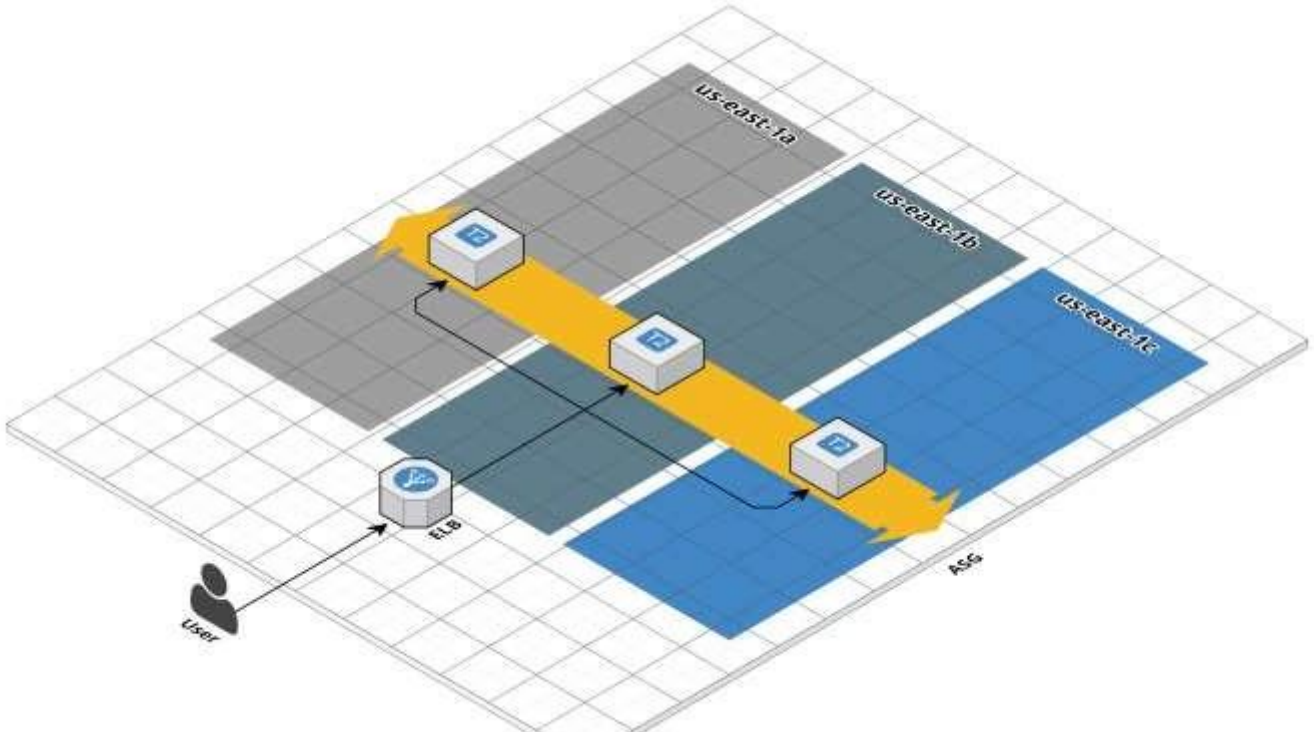
# OK, I have a server running!



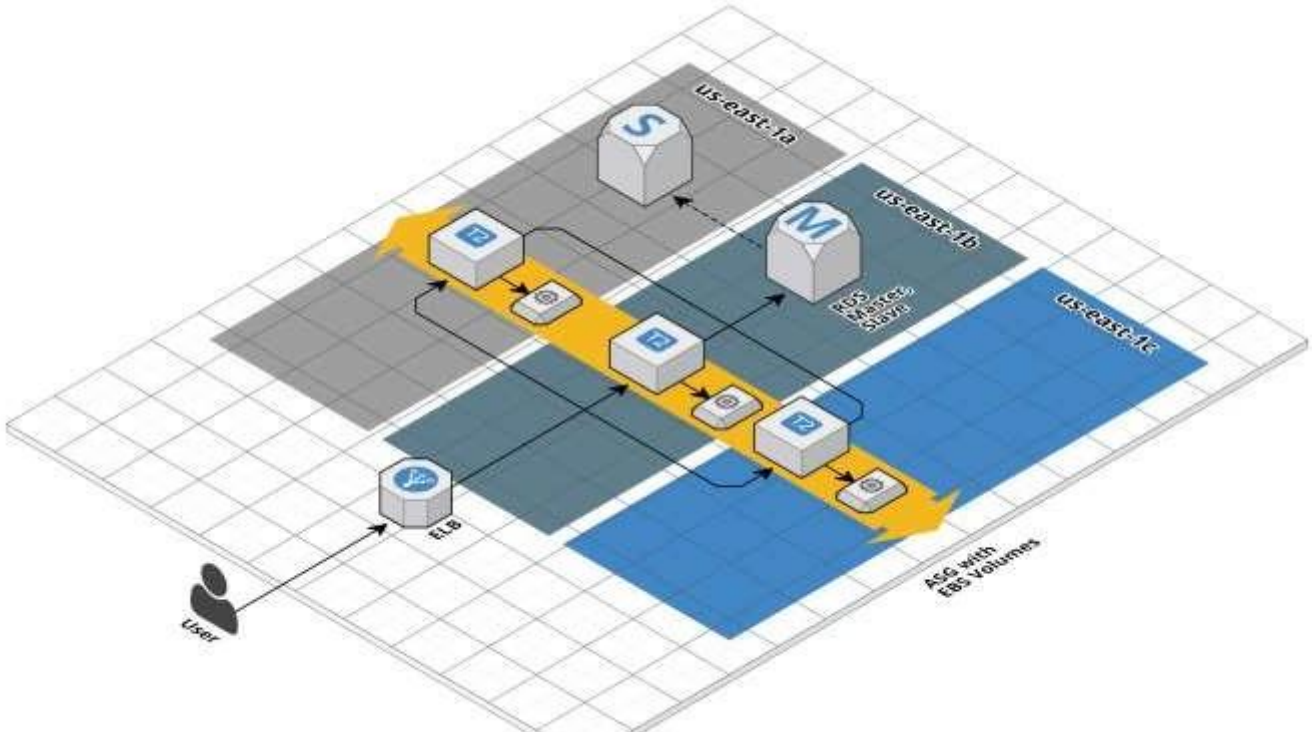
# What else do I need?



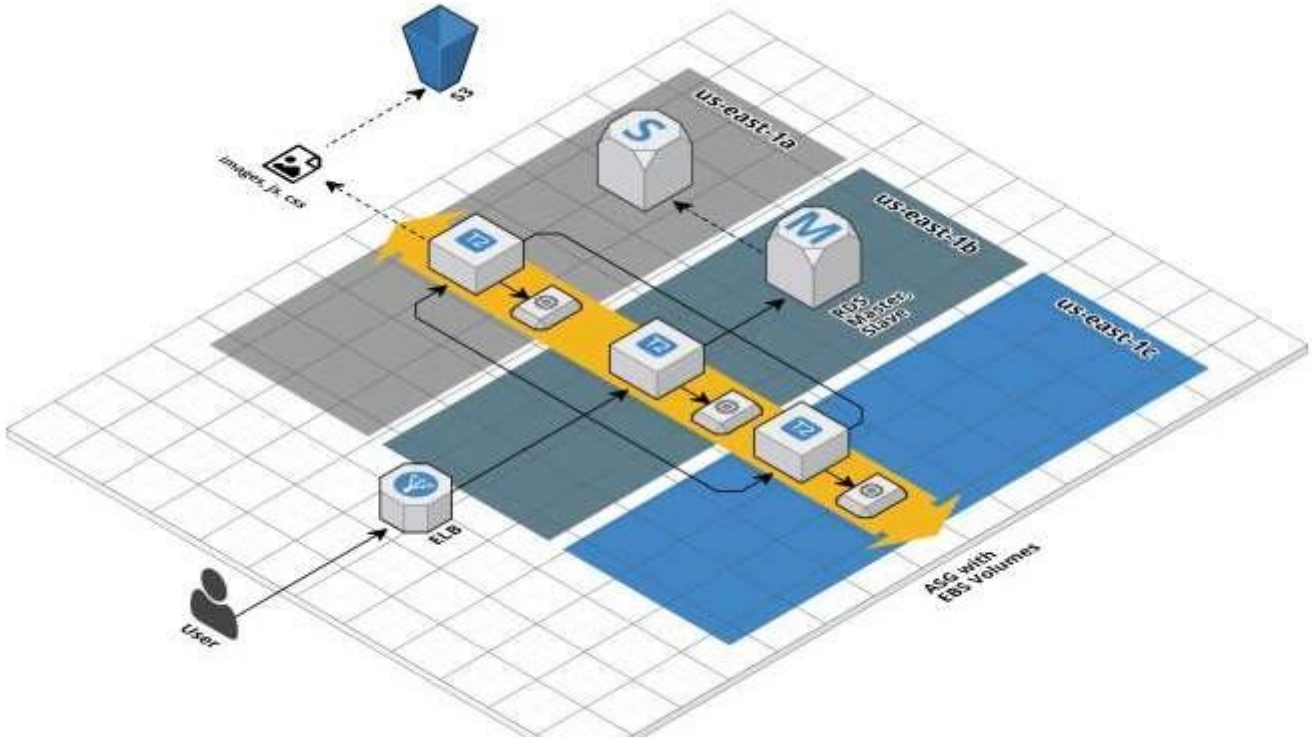

# Well, you probably want more than one server for high availability



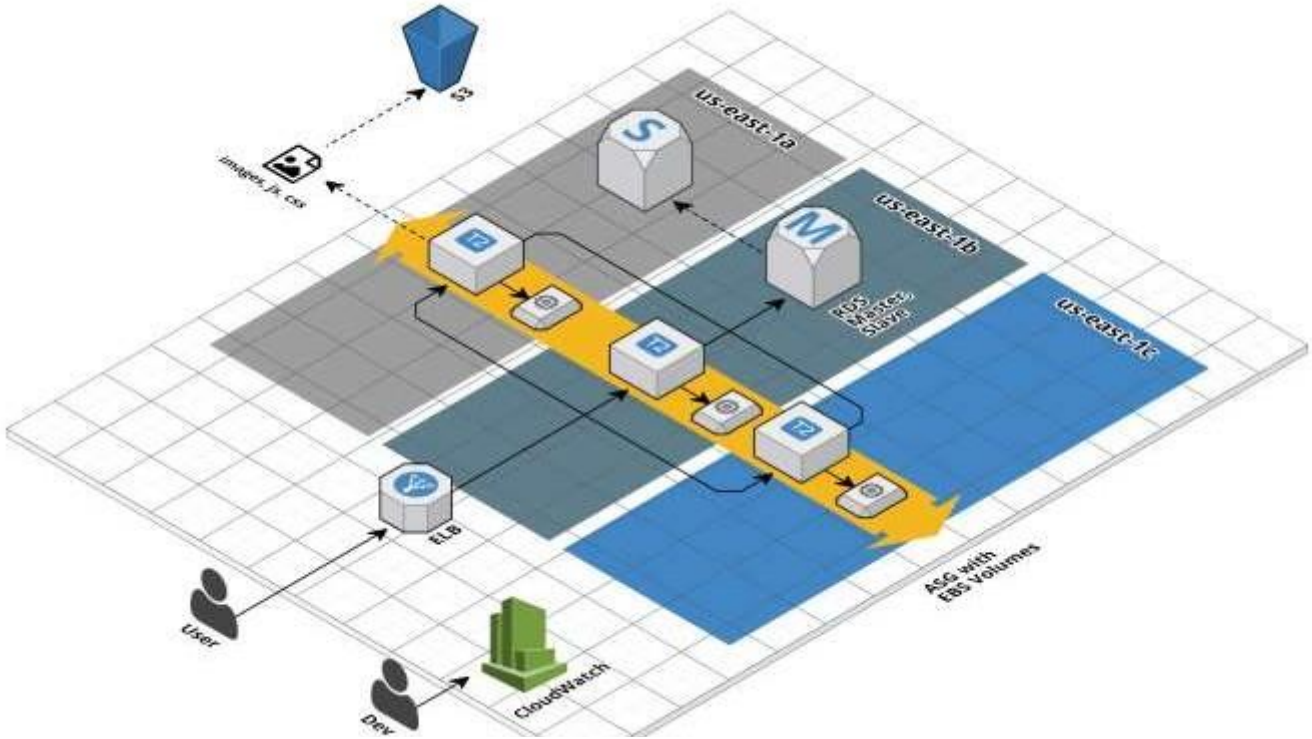

# And a load balancer to distribute traffic across those servers



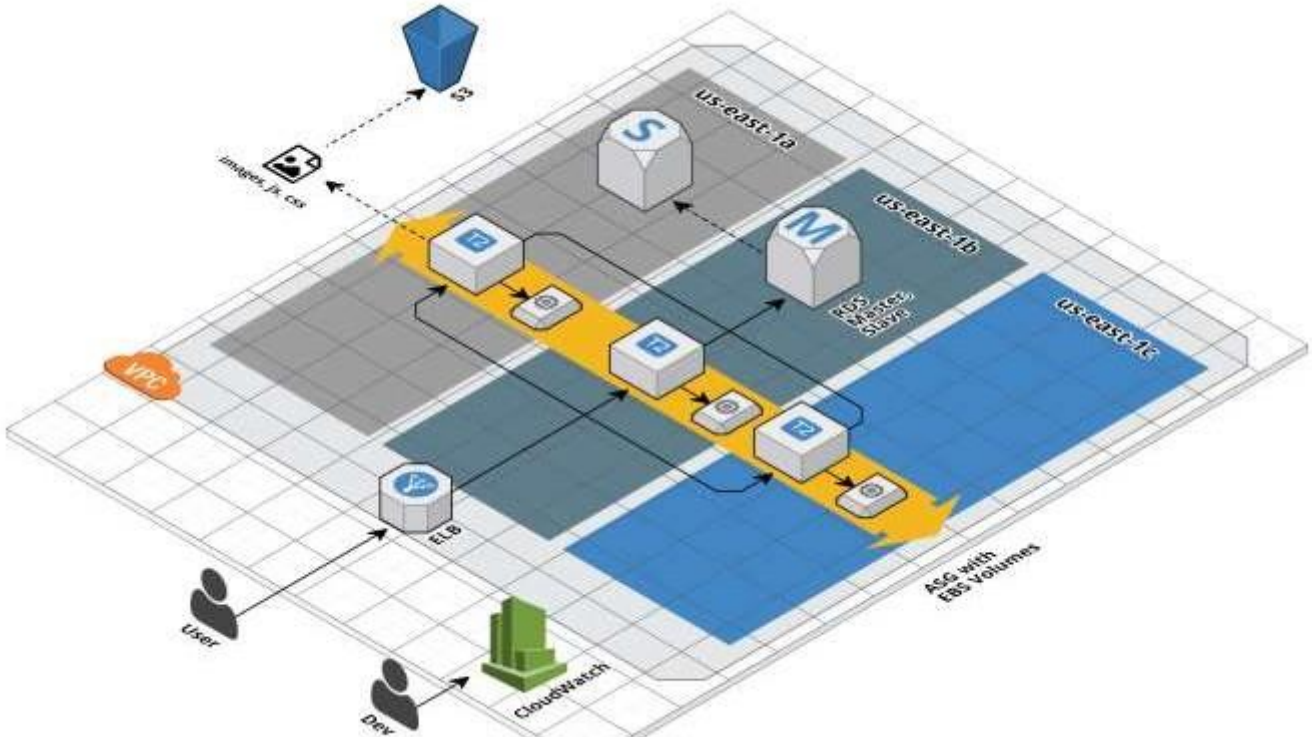

# And EBS Volumes and RDS databases to store all of your data



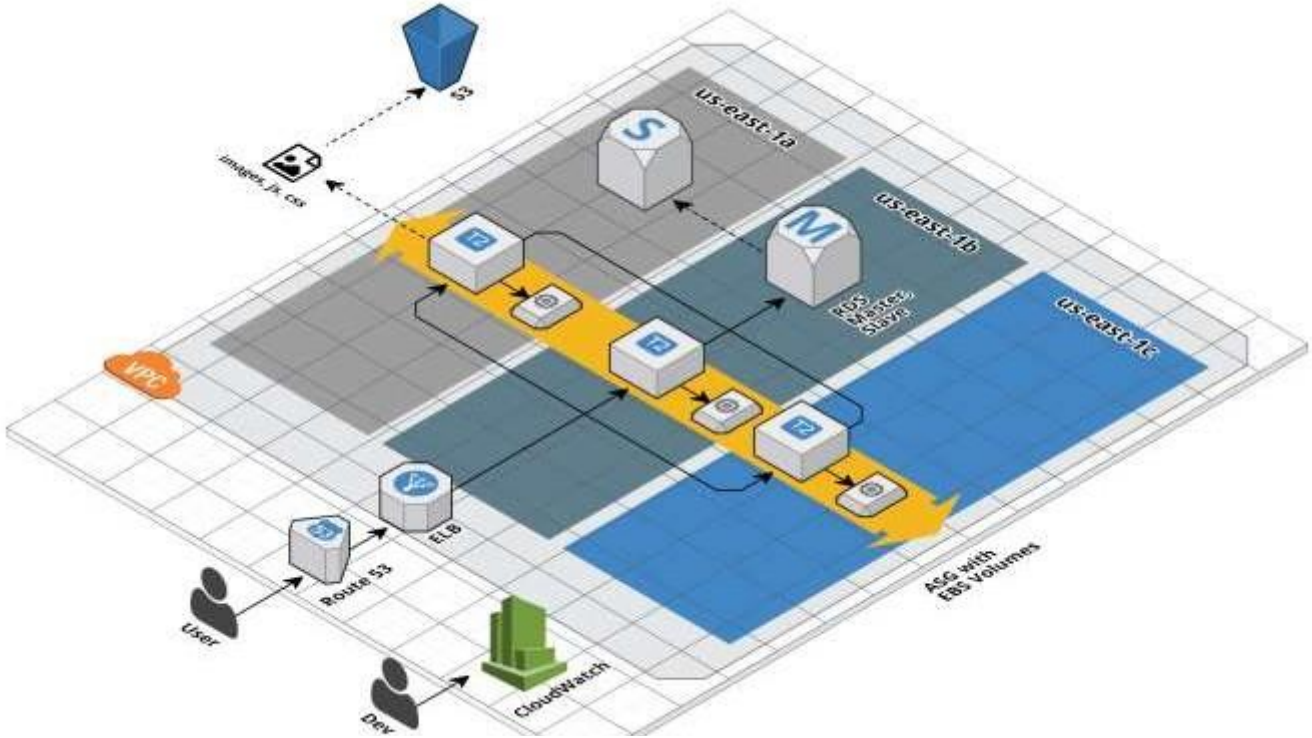

# You'll need an S3 bucket for files



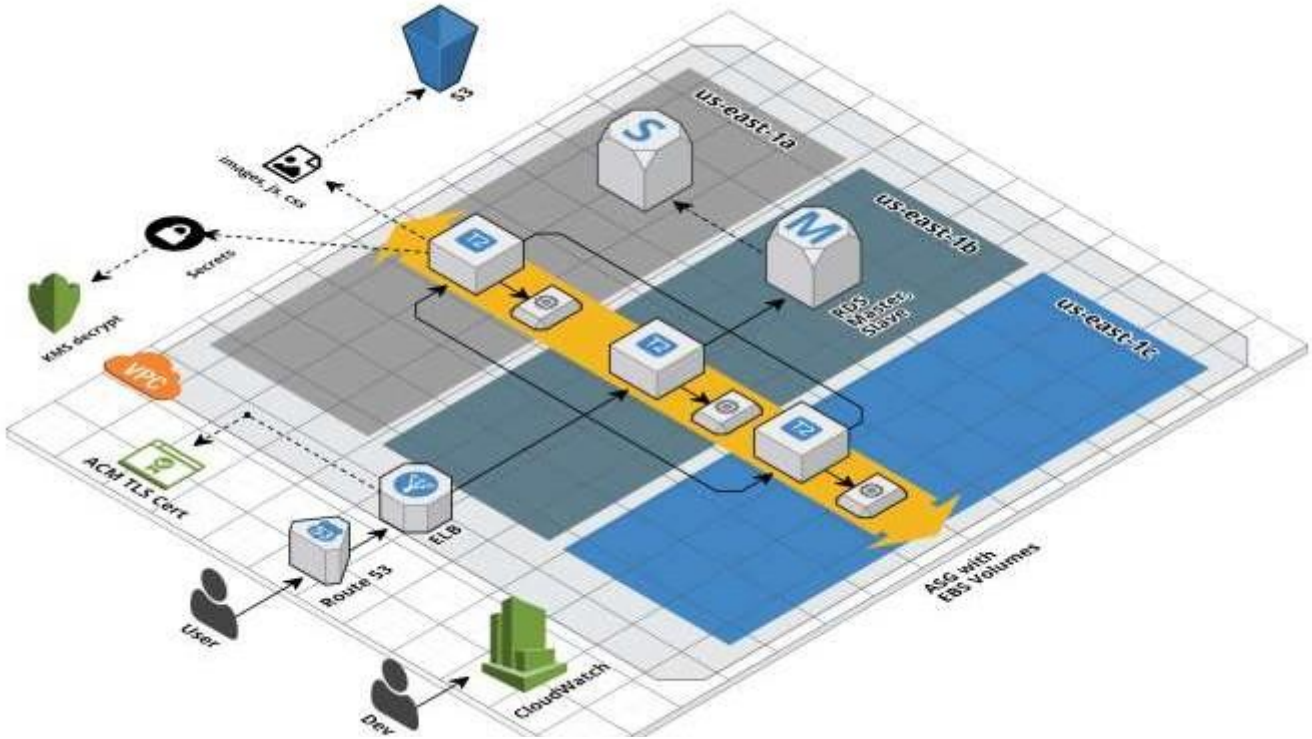

# CloudWatch for monitoring




# Don’t forget the VPC, subnets, route tables, NAT gateways

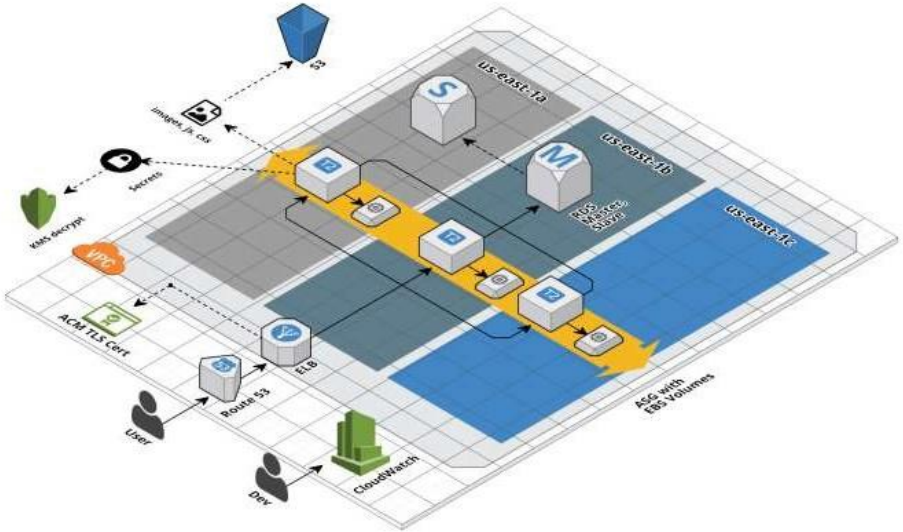



# Route 53 for DNS

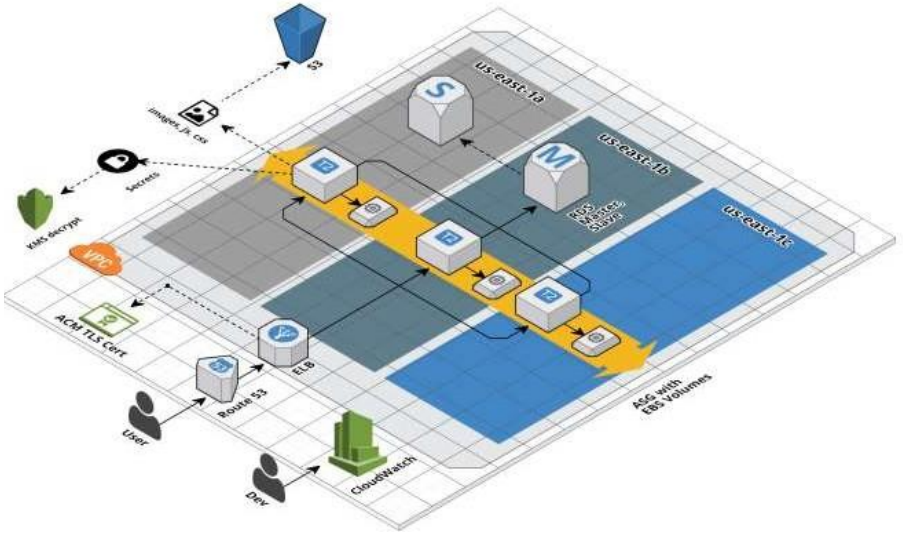



# ACM for SSL/TLS certs and KMS to encrypt / decrypt secrets

stage




prod




**And you need all of that in separate environments for stage and prod**

stage






prod

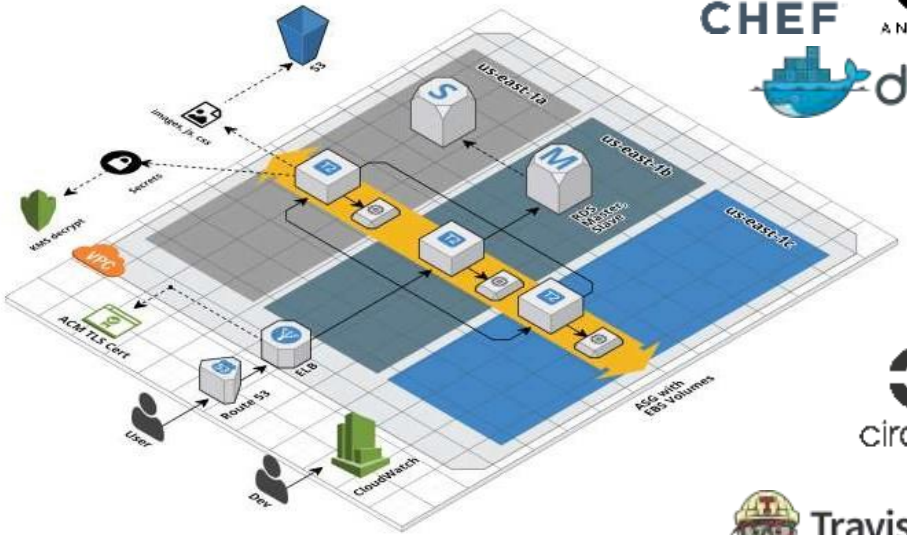






# Plus DevOps tooling to manage it all

stage




Terraform

CHEF

ANSIBLE

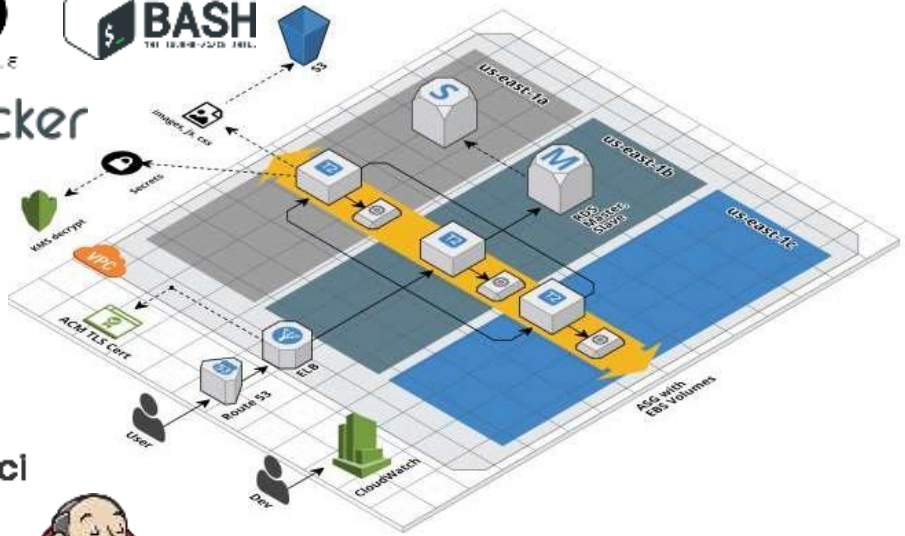
BASH

docker

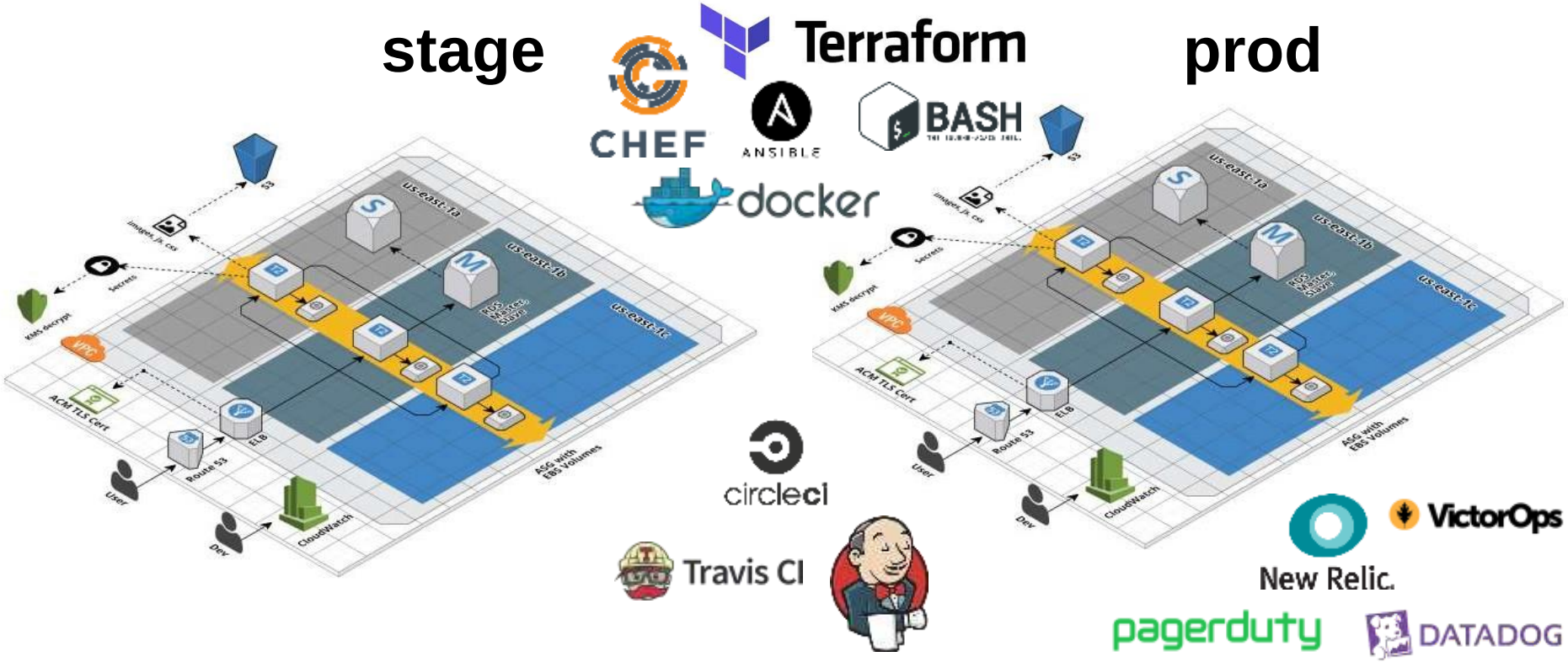
circleci

Travis CI

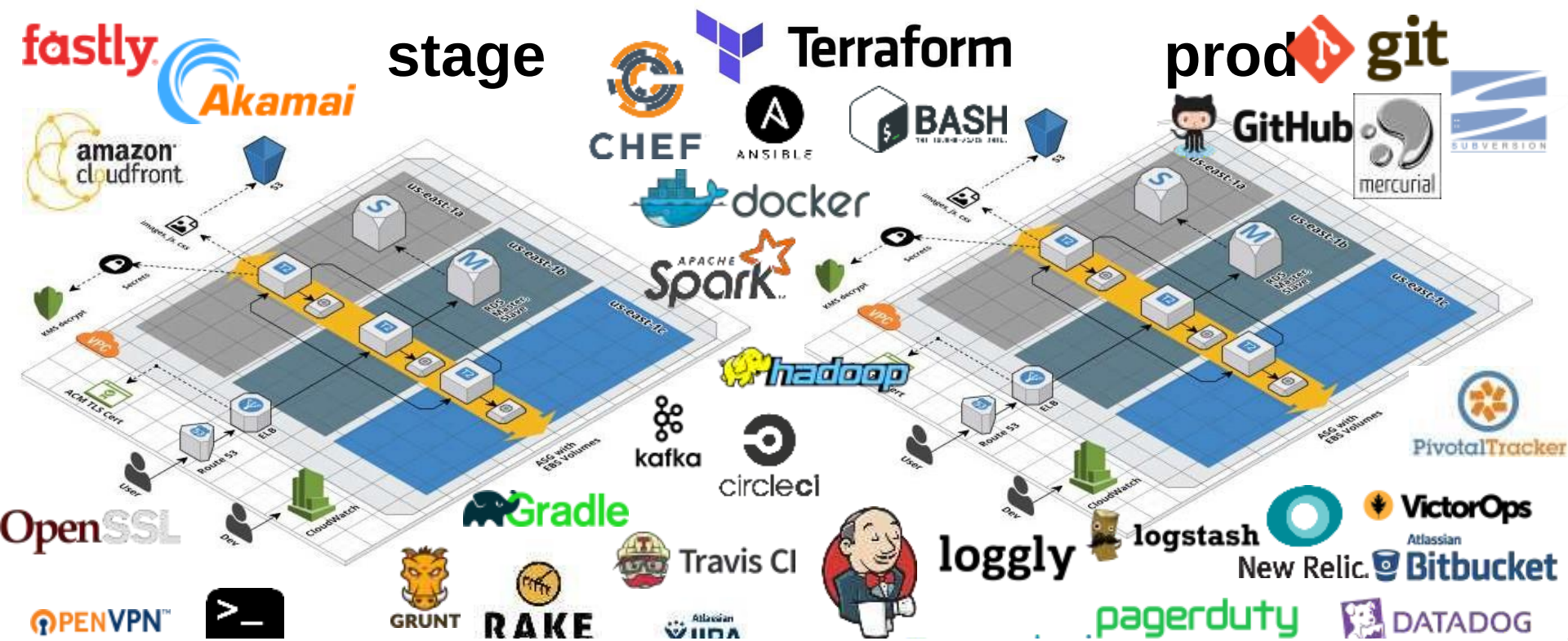
prod




# And a CI server to test it all



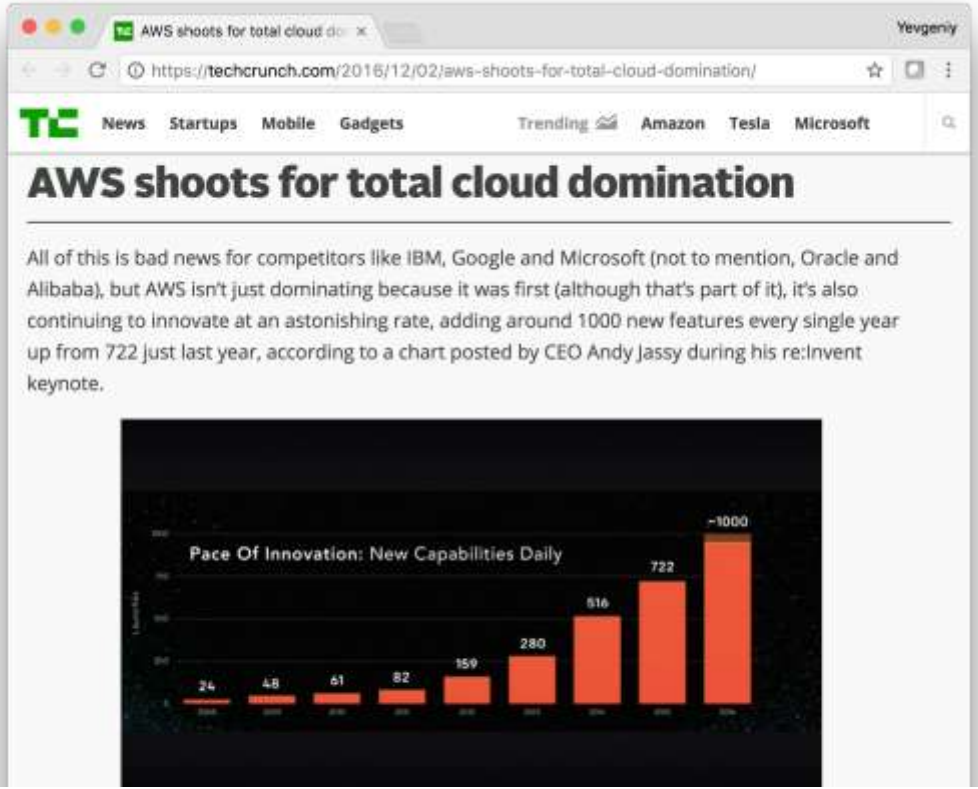

# Plus alerts and on-call rotation to notify you when it all breaks


fastly
Akamai
stage
Terraform
prod
git
amazon cloudfront
CHEF
ANSIBLE
BASH
GitHub
mercurial
SUBVERSION
docker
APACHE Spark
hadoop
kafka
circleci
PivotalTracker
OpenSSL
Gradle
Travis CI
loggly
logstash
VictorOps
New Relic.
Atlassian Bitbucket
OPENVPN
GRUNT
RAKE
pagerduty
DATADOG
us-east-1a
us-east-1b
us-east-1c
ELB
Route 53
CloudWatch
User
Dev
ASG with EBS Volumes
ACM TLS Cert
KMS decrypt
Secrets
VPC

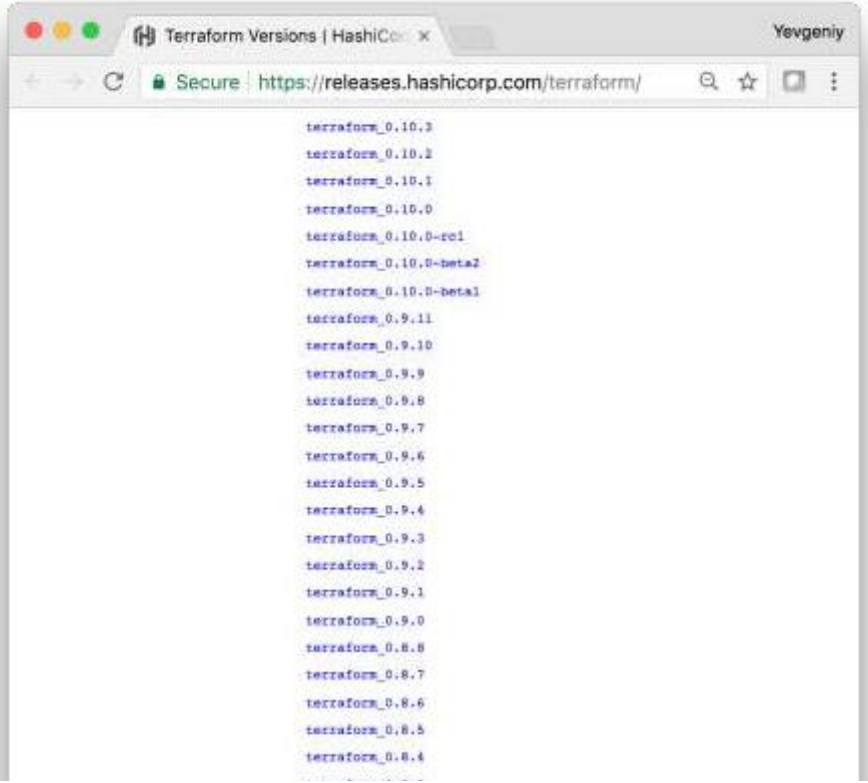
# And also…

FFFFFFF
FFFFFFF
FFFFFF
FFFUU
UUUU
UUUU
UUUU
UUUU
UUUU-

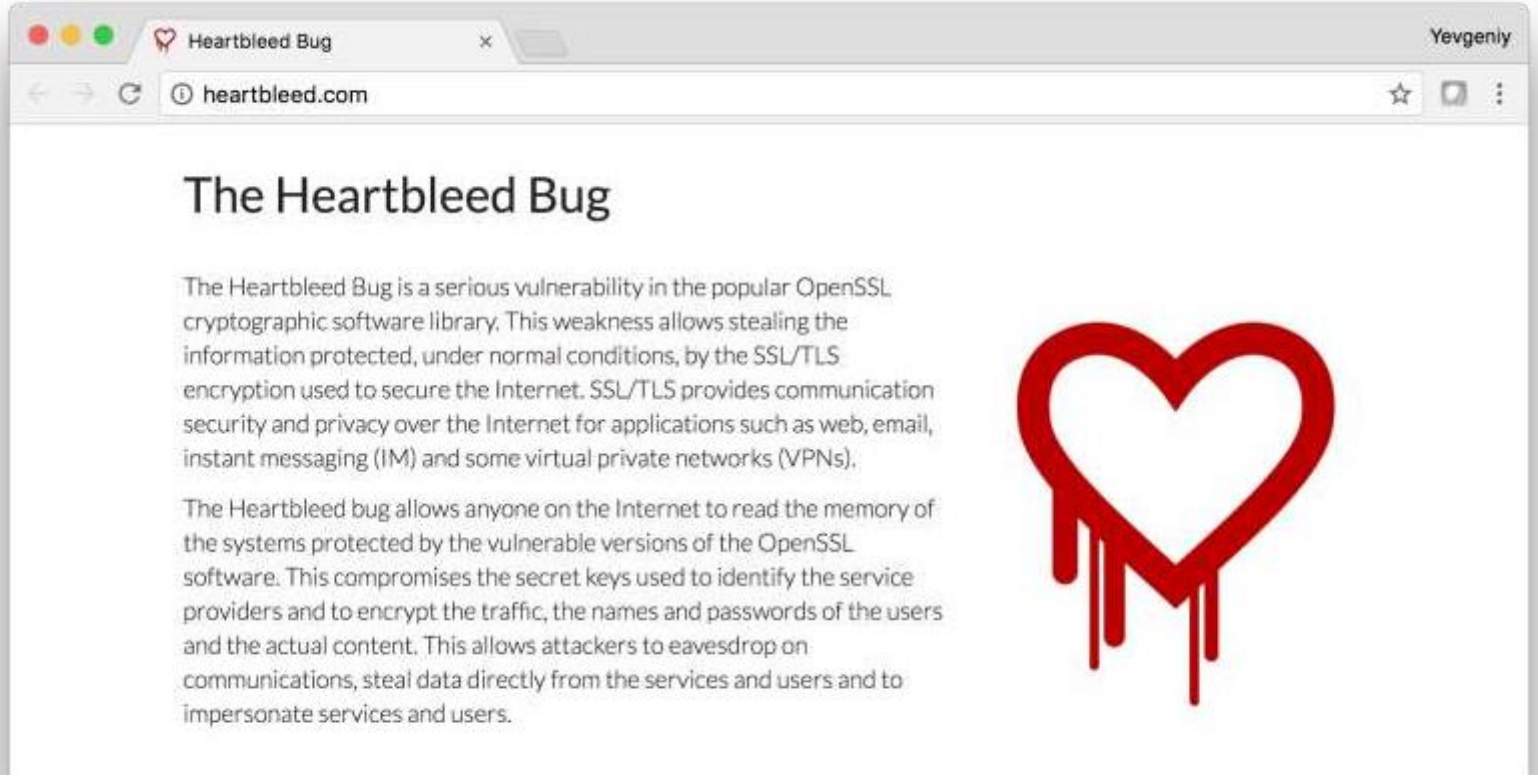
**And you have to maintain it all. Forever.**



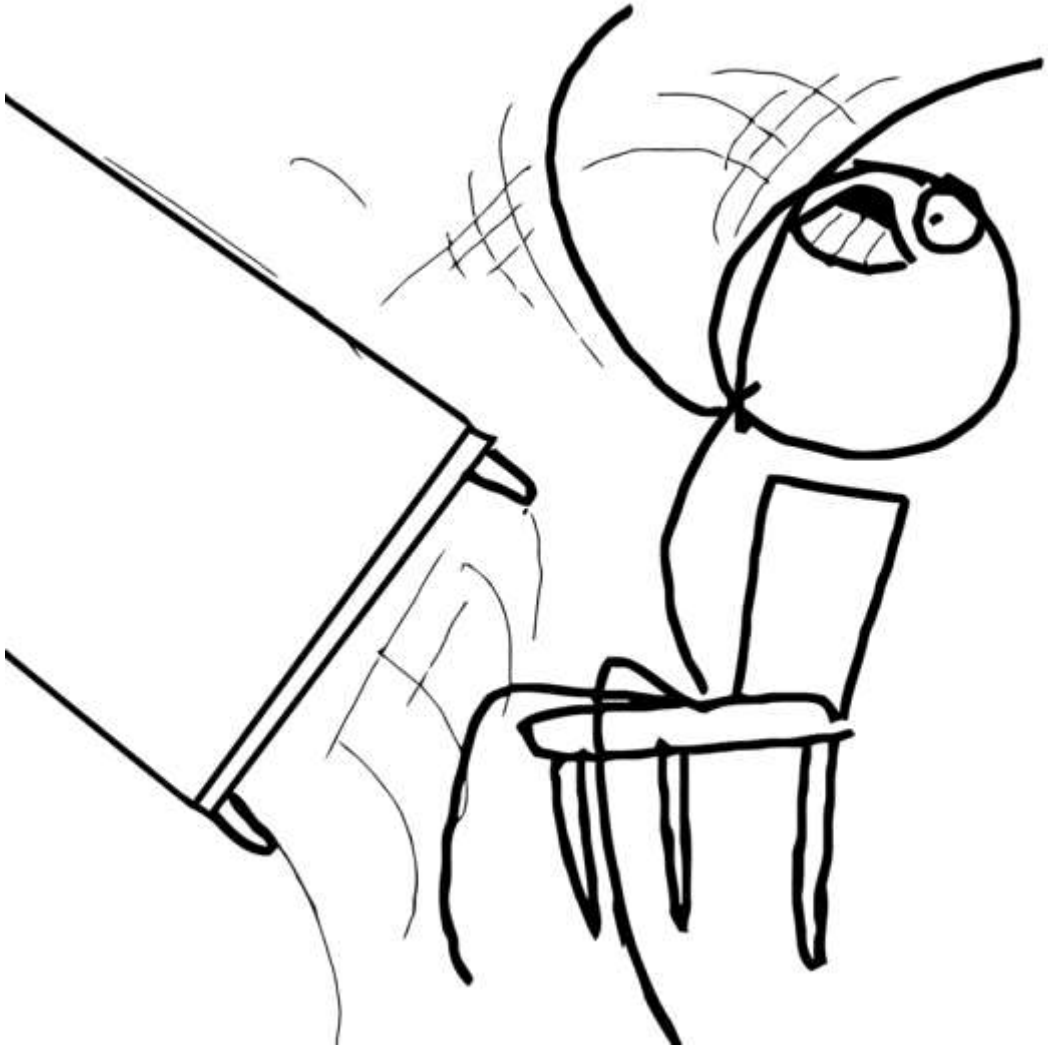

# AWS: 1,000 new releases in 2021




# Terraform: release every ~2 weeks




# Security vulnerabilities: daily



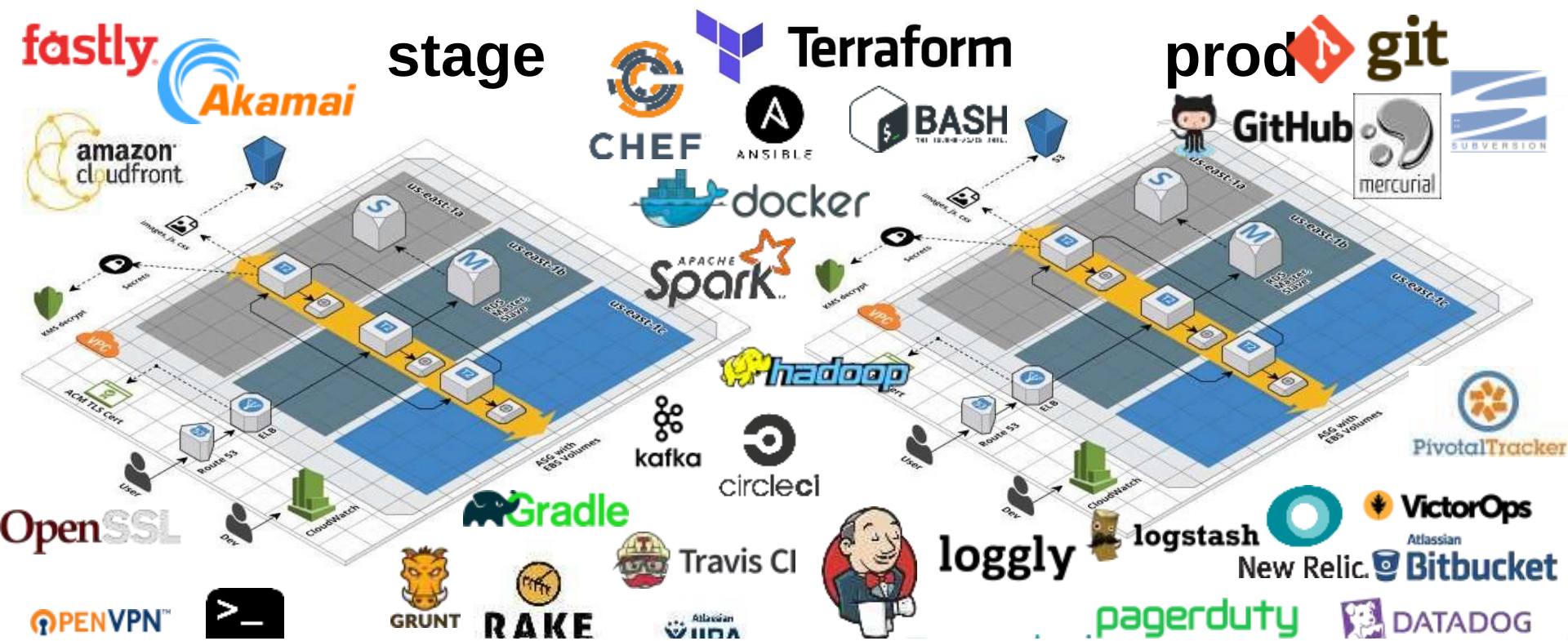
# There’s a better way to deploy and manage infrastructure:

# TERRAFORM MODULES

# TERRAFORM MODULES

*Reusable, composable, battle-tested infrastructure code*

# I’ll show you
# How  Terraform Modules work?



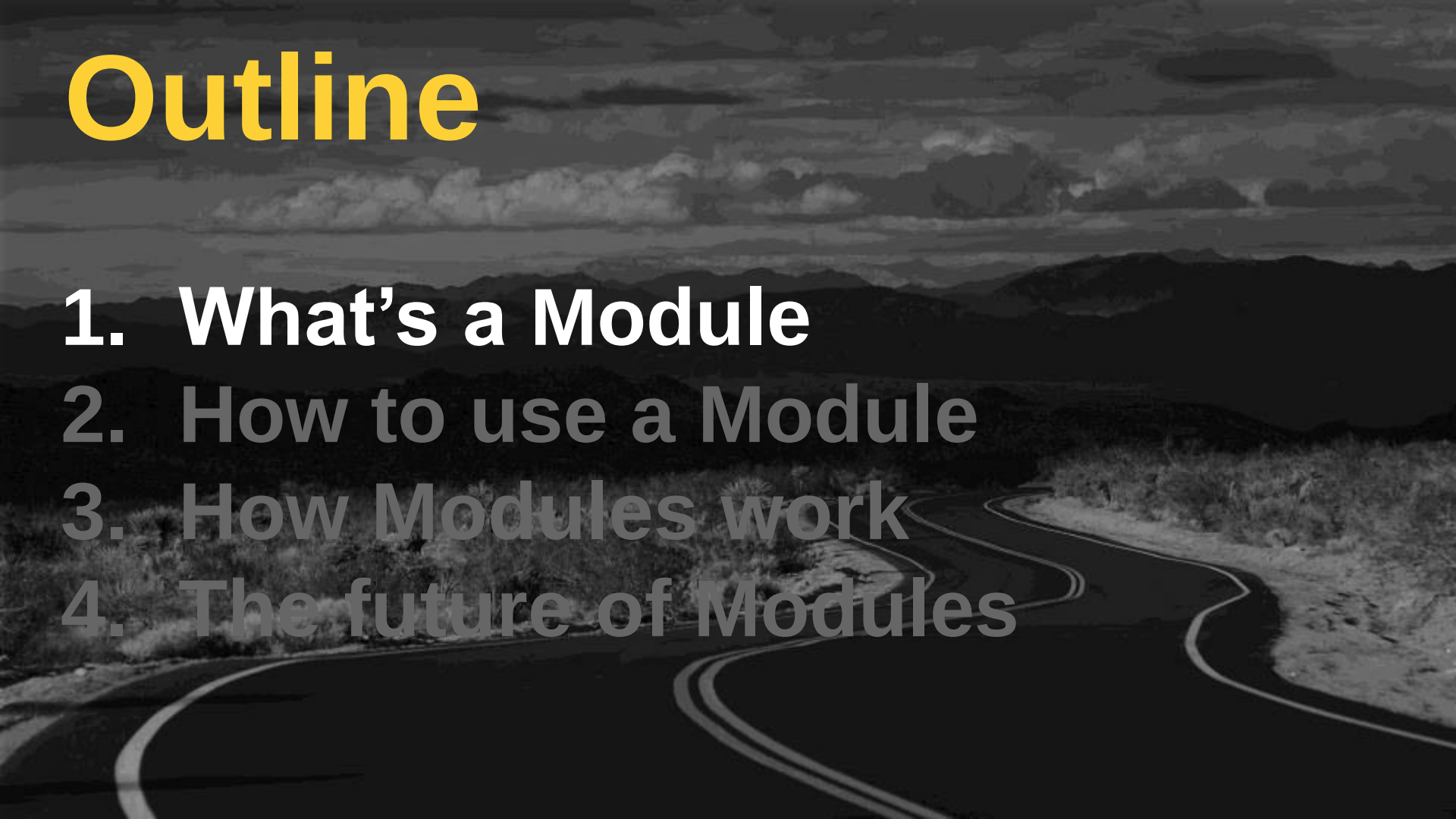

# And how they will allow you to do all of this…

```
> terraform init <..>

> terraform apply
```

# In just a few simple commands

# Outline

1. What's a Module
2. How to use a Module
3. How Modules work
4. The future of Modules



# Outline

1. What's a Module
2. How to use a Module
3. How Modules work
4. The future of Modules

# Terraform Modules

Code reuse

Remote or local source

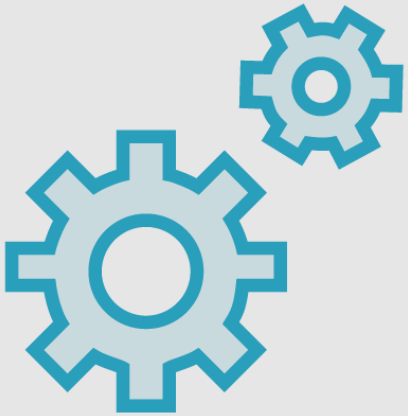
Terraform evaluation

Mini-Terraform configuration

Multiple instances (no count)

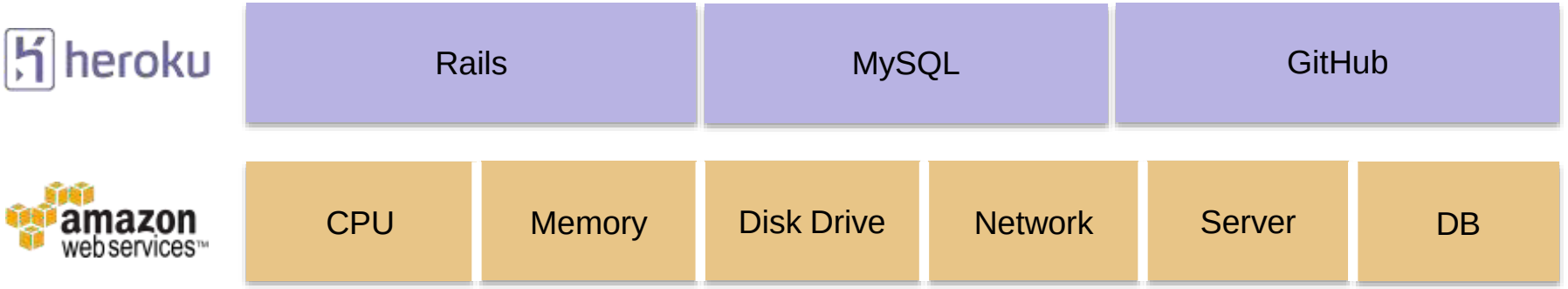
# Module Components

Variables

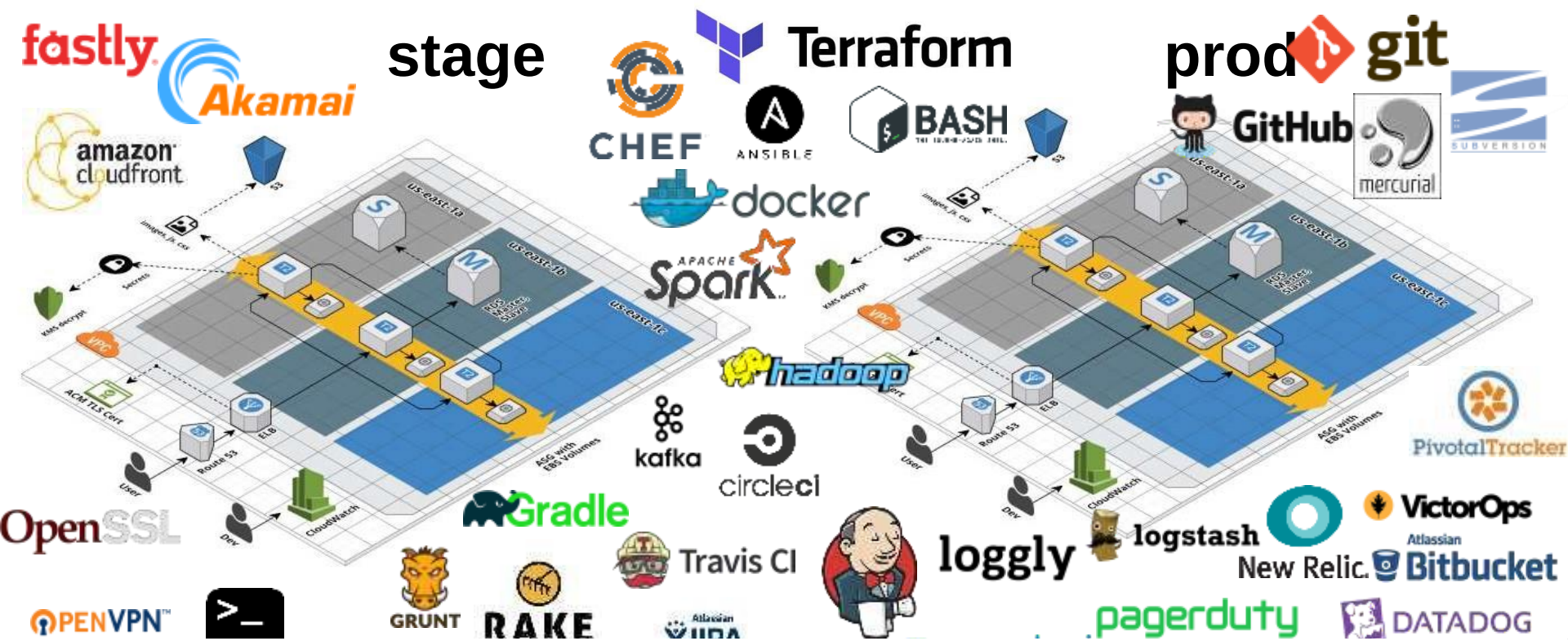


Resources

Outputs




# Platform as a Service (PaaS): e.g., Heroku, Docker Cloud, Engine Yard




**But that requires dealing with this…**

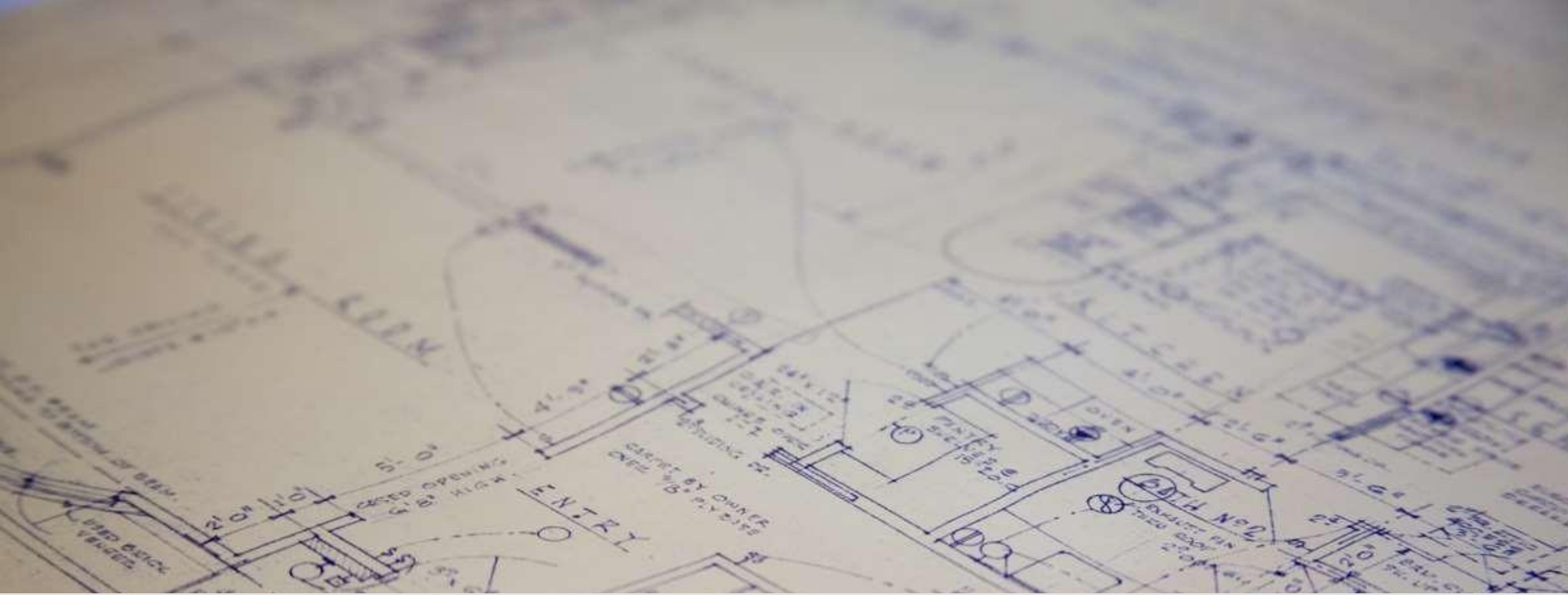

# Terraform is a tool for defining and managing infrastructure as code

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

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

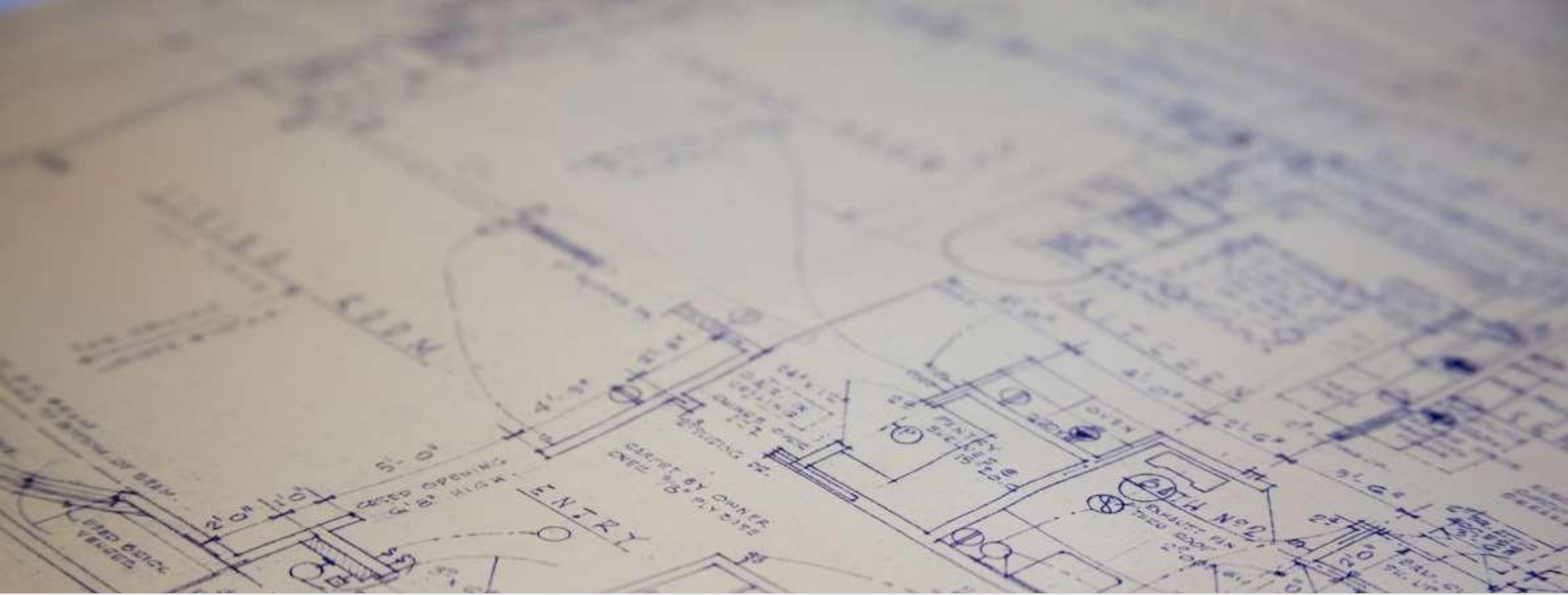
# Example: Terraform code to deploy a server in AWS

```
> 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.
```

**Run `terraform apply` to deploy the server**




# Terraform supports modules




# They are like reusable blueprints for your infrastructure

```
resource "aws_autoscaling_group" "example" {
  name     = "${var.name}-service"
  min_size = "${var.num_instances}"
  max_size = "${var.num_instances}"
}

resource "aws_launch_configuration" "example" {
  image_id      = "${var.image_id}"
  instance_type = "${var.instance_type}"
  root_block_device {
}
```

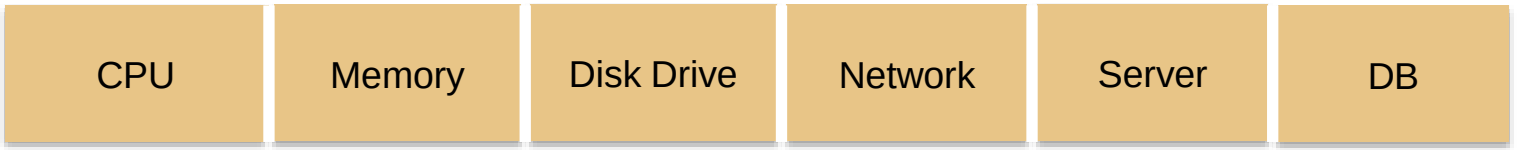
# Example: create a module to deploy a microservice

```
module "service_foo" {
  source          = "/modules/microservice"
  image_id        = "ami-123asd1"
  num_instances = 3
}
```

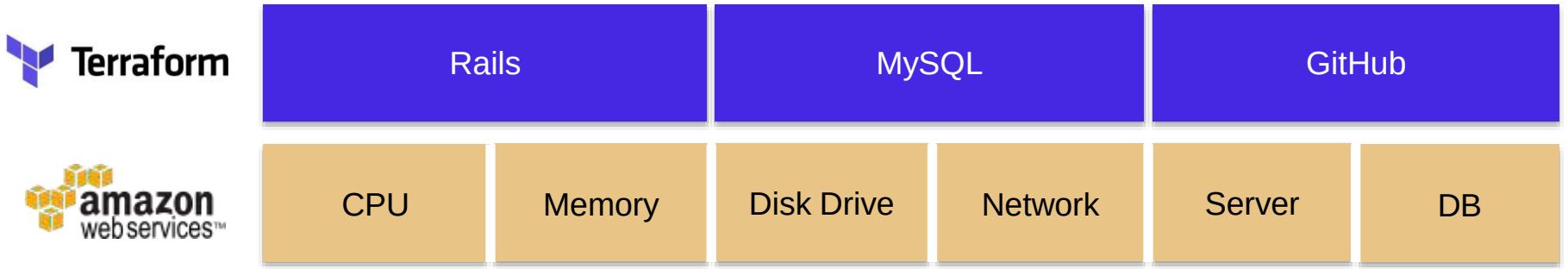
**Now you can use the module to deploy one microservice**

```
module "service_foo" {
  source        = "/modules/microservice"
  image_id      = "ami-123asd1"
  num_instances = 3
}
module "service_bar" {
  source        = "/modules/microservice"
  image_id      = "ami-f2bb05ln"
  num_instances = 6
}
```

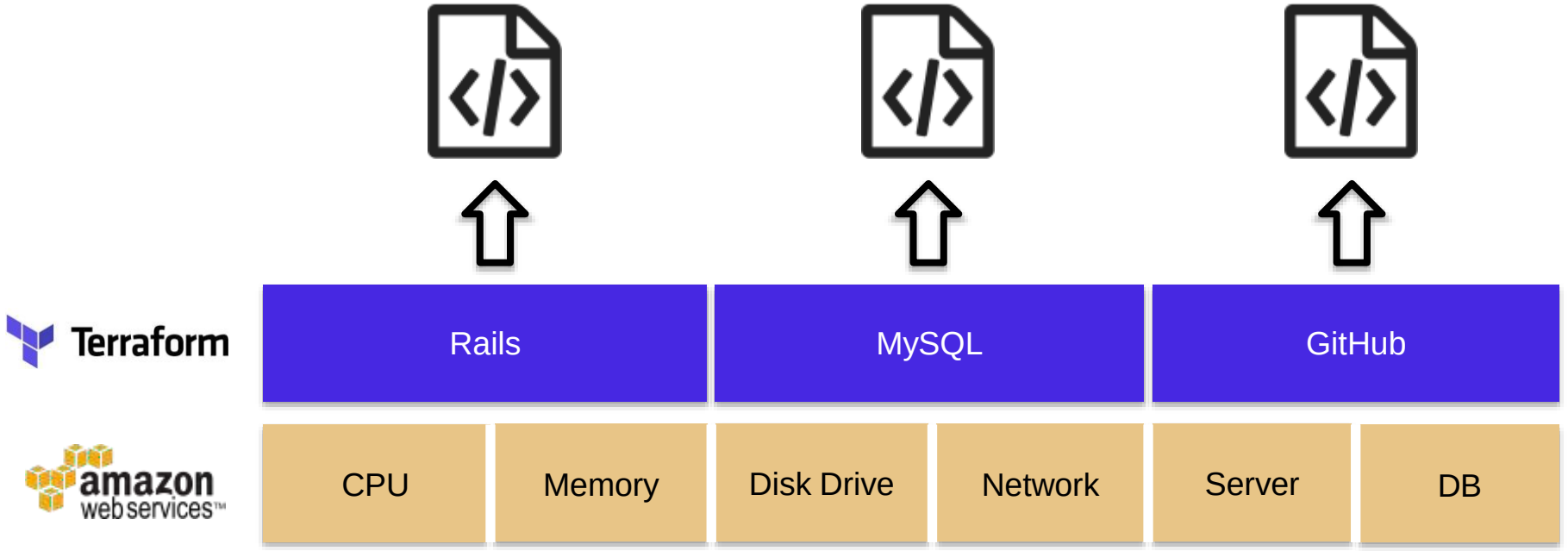
# Or multiple microservices


# Modules allow you to use your favorite IaaS provider…

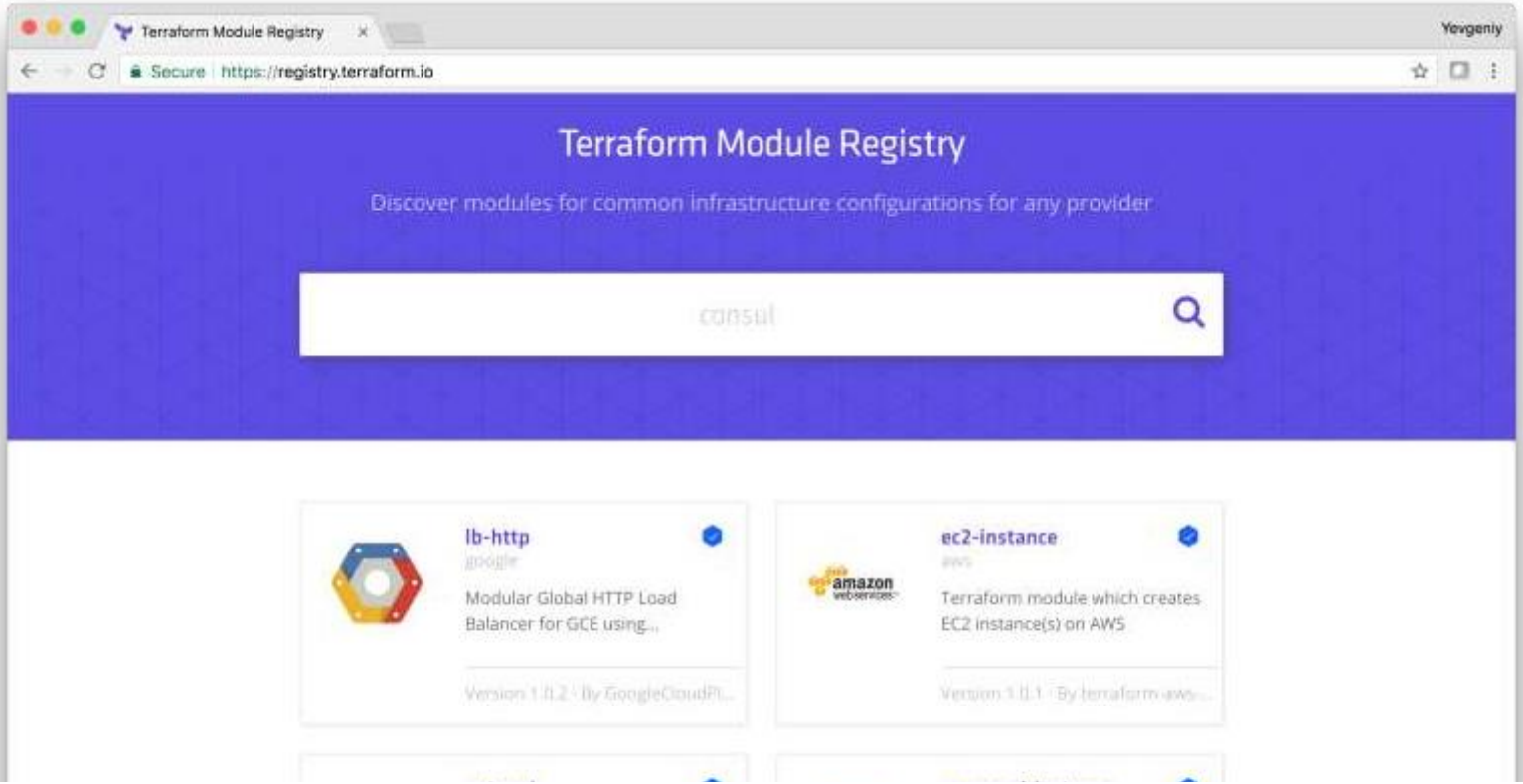



# With easy-to-use, high-level abstractions, like a PaaS

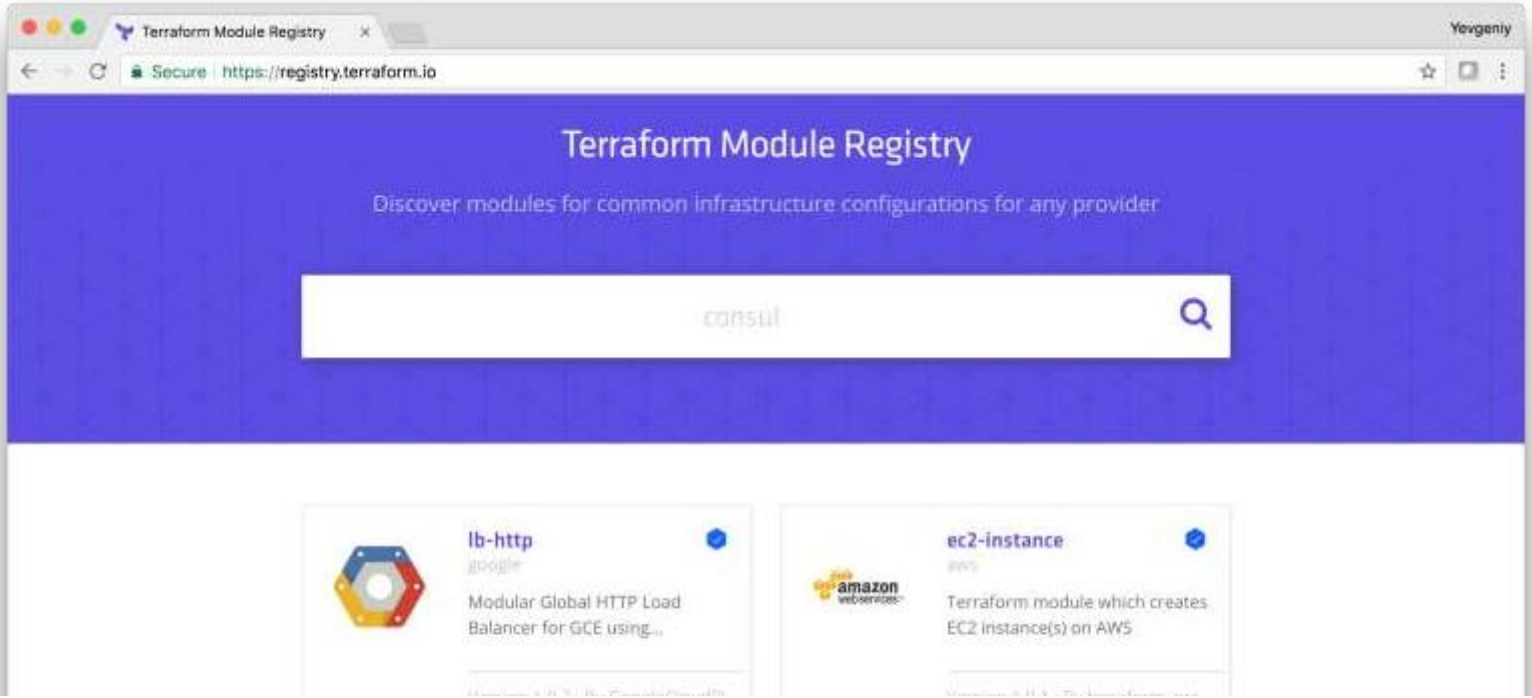



**But since you have all the code, you still have full control!**

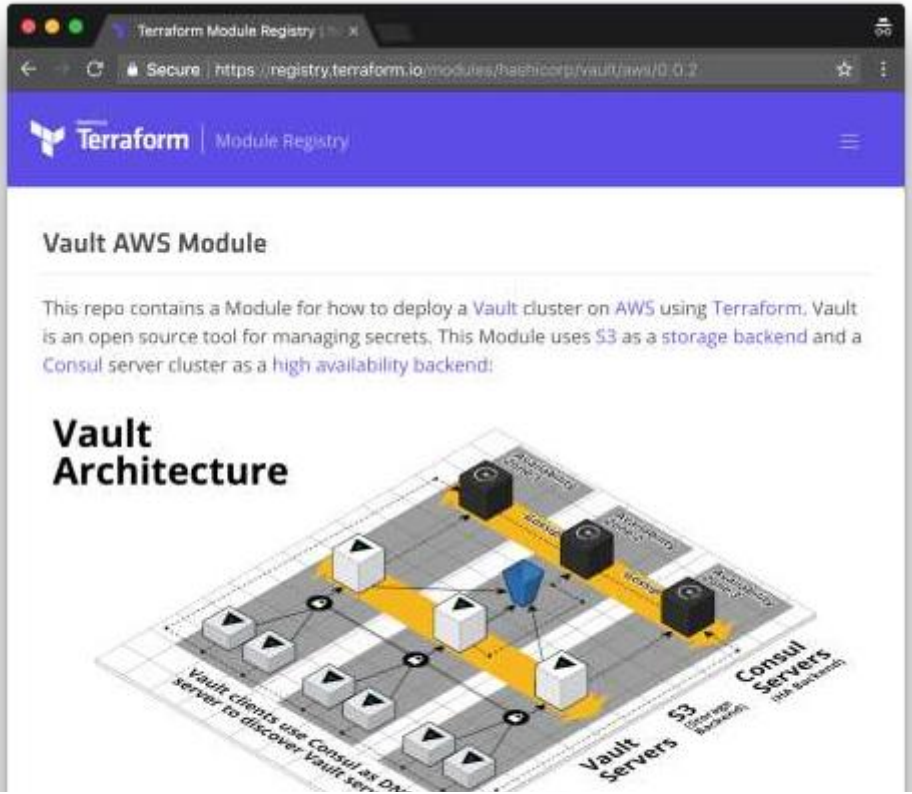
**Best of all: code can be shared and re-used!**



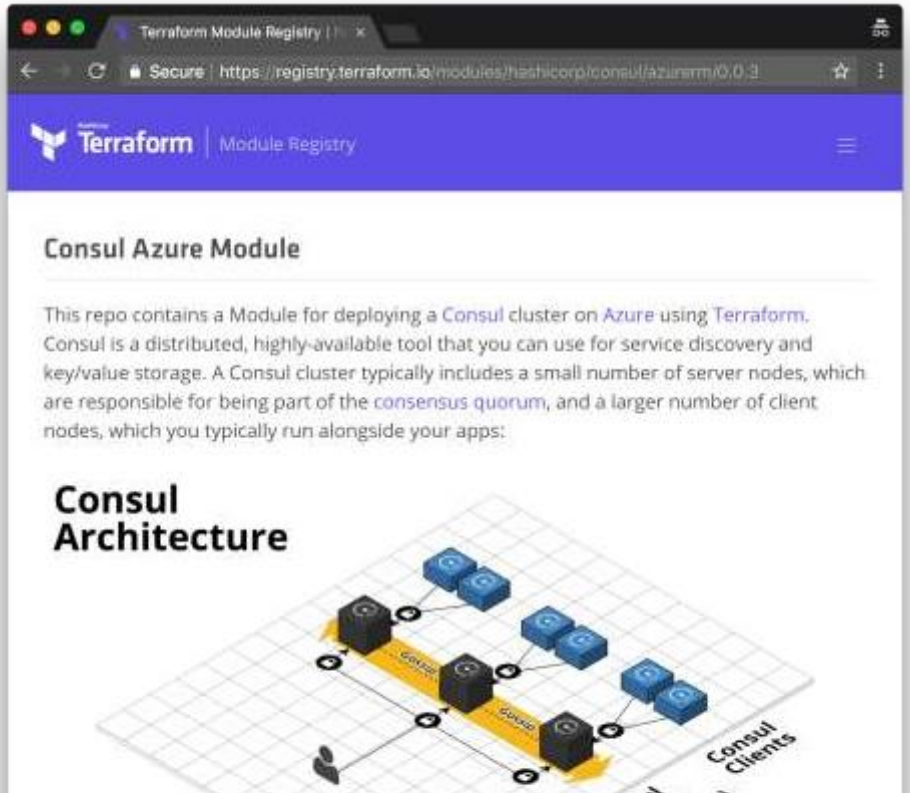

# The Terraform Module Registry



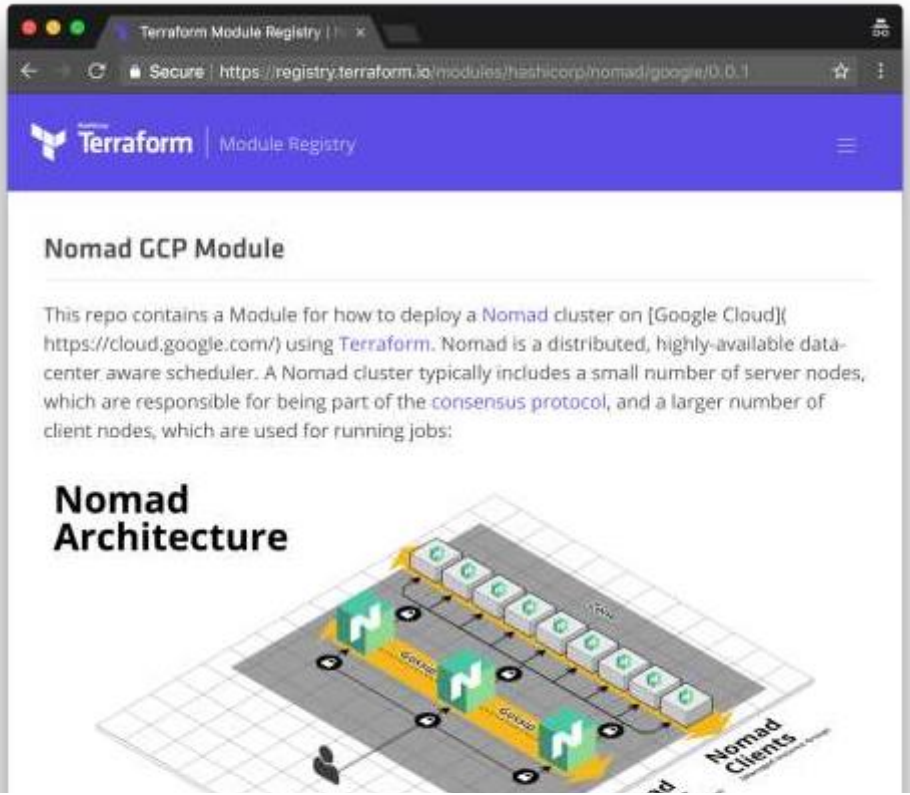

# A collection of reusable, verified, supported Modules




# Example: Vault Module for AWS



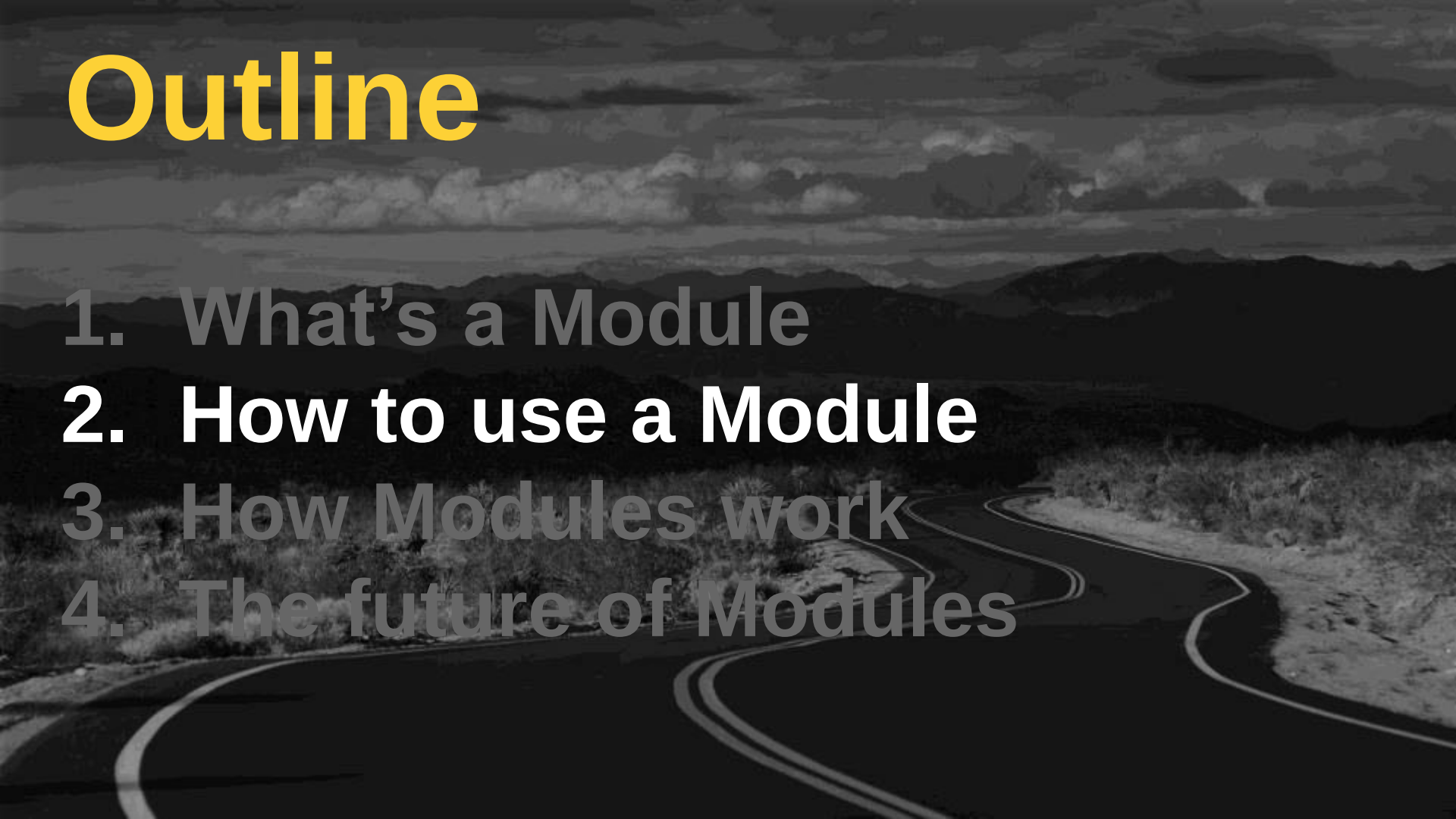

# Example: Consul for Azure




# Example: Nomad for GCP

# Outline

1. What's a Module
2. How to use a Module
3. How Modules work
4. The future of Modules




# Imagine you wanted to deploy Vault

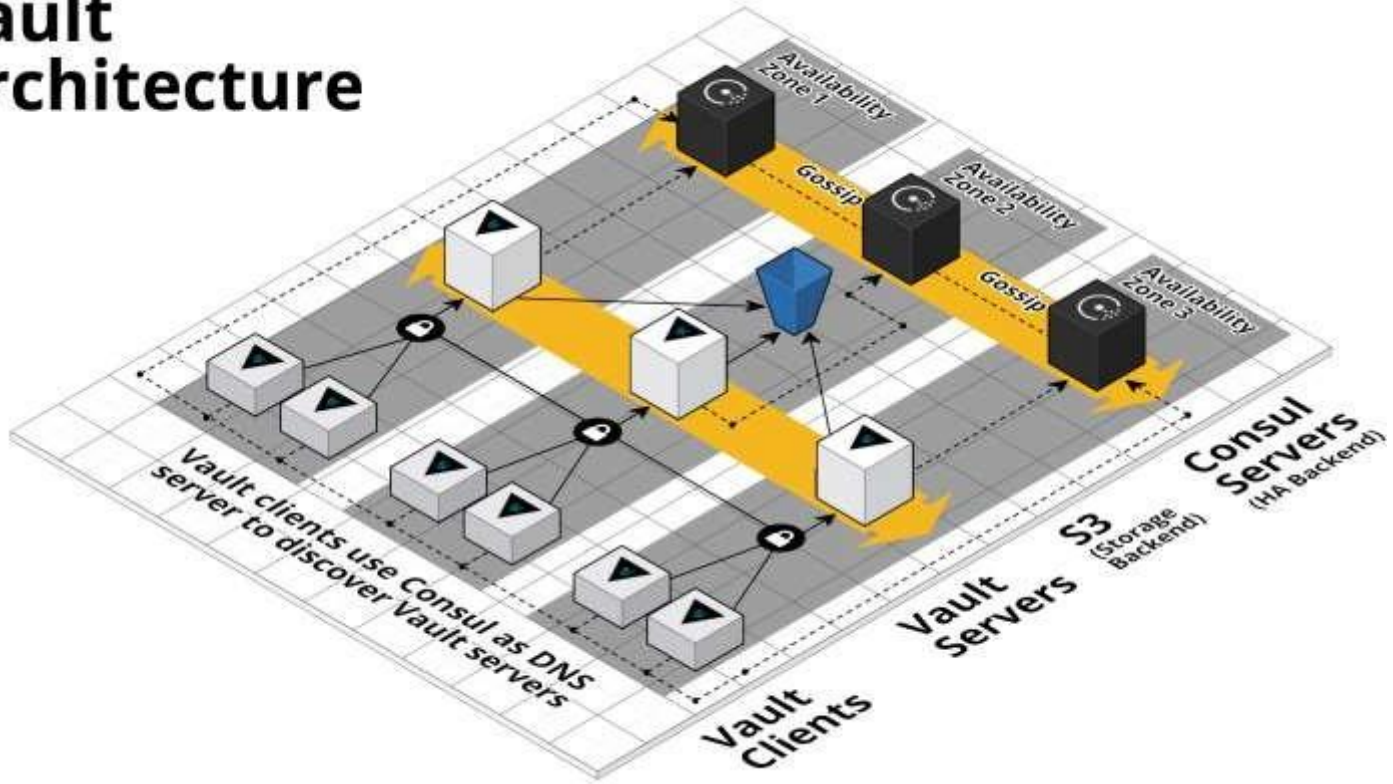
# The old way:

1. Open up the Vault docs
2. Deploy a few servers
3. Install Vault
4. Install supervisord
5. Configure mlock
6. Generate self-signed TLS cert
7. Create Vault config file
8. Create an S3 bucket as storage backend
9. Figure out IAM policies for the S3 bucket
10. Tinker with security group rules
11. Figure out IP addresses to bind to and advertise
12. Fight with Vault for hours because it won’t accept your TLS cert
13. Regenerate cert with RSA encryption
14. Update OS certificate store to accept self-signed certs

# The new way:

```
> terraform init hashicorp/vault/aws

> terraform apply
```

# Vault Architecture




**And now you have this deployed**

# As simple as a PaaS!

```
> tree
.
├── README.md
├── main.tf
├── outputs.tf
├── packer
├── user-data
└── variables.tf
```

**But you also have all the code. Feel free to edit it!**

```
module "vault_cluster" {
    source       = "hashicorp/vault/aws"
    cluster_name = "example-vault-cluster"
    cluster_size = 3
    vpc_id       = "${data.aws_vpc.default.id}"
    subnet_ids   = "${data.aws_subnets.default.ids}"
}
module "consul_cluster" {
    source       = "hashicorp/consul/aws"
    cluster_name = "example-consul-cluster"
```

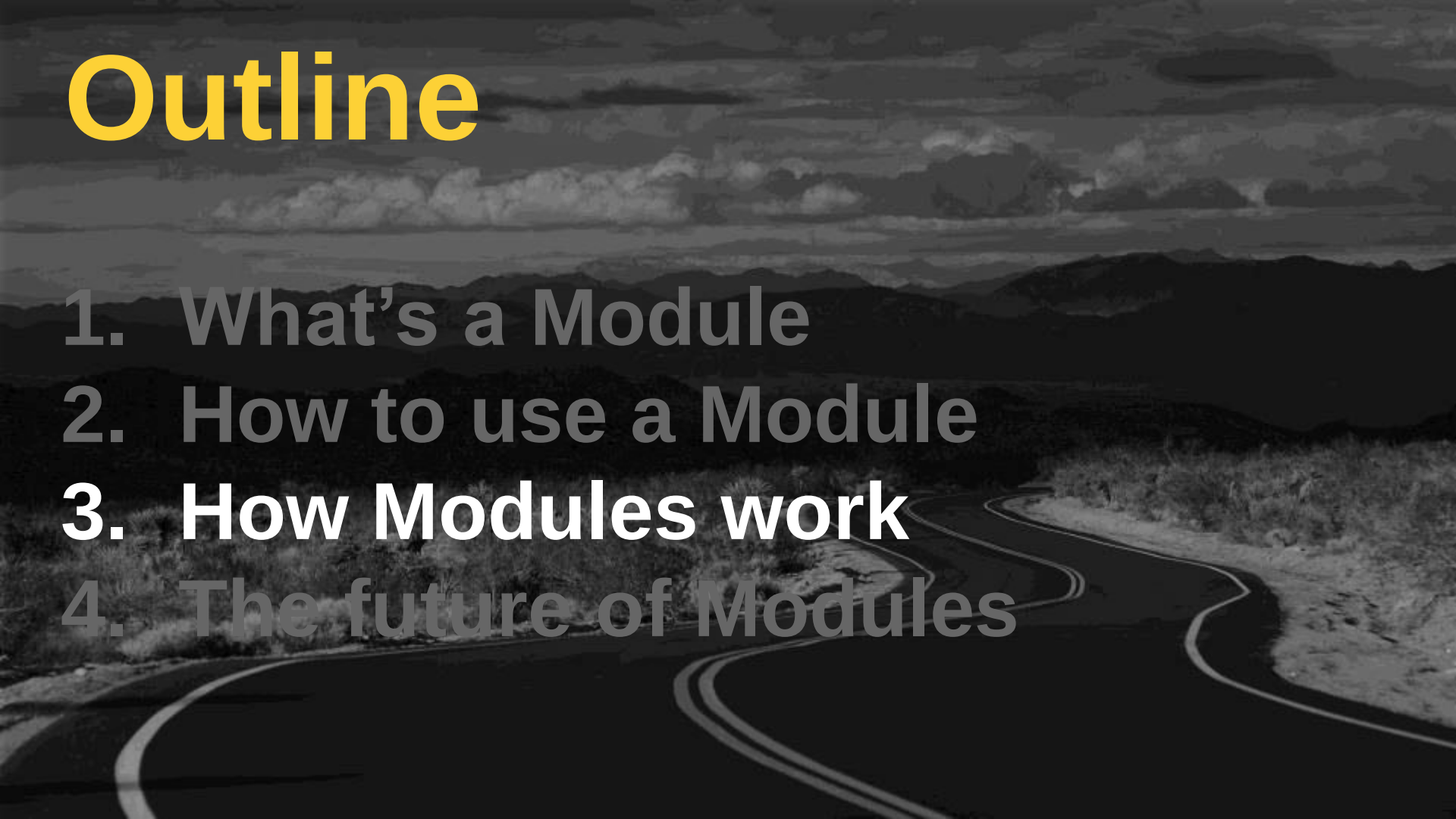
# Example: modify main.tf

```
> terraform apply
```

# Run apply when you're done!

# Outline

1. What's a Module
2. How to use a Module
3. How Modules work
4. The future of Modules



```
def add(x, y):
  return x + y
```

# Most programming languages support functions

```
def add(x, y):
  return x + y
```

# The function has a name

```
def add(x, y):
  return x + y
```

# It can take in inputs

```
def add(x, y):
  return x + y
```

# And it can return outputs

```
def add(x, y):
  return x + y


add(3, 5)
add(10, 35)
add(-45, 6)
```

# Key idea: code reuse

```
def add(x, y):
  return x + y

assert add(3, 5)   == 8
assert add(10, 35) == 45
assert add(-45, 6) == -39
```

# Key idea: testing

```
def add(x, y):
  return x + y

def sub(x, y):
  return x - y

sub(add(5, 3), add(4, 7))
```

# Key idea: composition

# Modules are Terraform’s equivalent of functions

# A simple module:

```
> tree minimal-module
.
├── main.tf
├── outputs.tf
├── variables.tf
└── README.md
```

# It’s just Terraform code in a folder!

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

variable "image_id" {
    description = "The ID of the AMI to run"
}

  variable   "port" {
       description = "The port to listen on for HTTP requests"
```

# The inputs are in variables.tf

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

# The outputs are in outputs.tf

```
resource "aws_autoscaling_group" "example" {
  name      = "${var.name}-service"
  min_size = "${var.num_instances}"
  max_size = "${var.num_instances}"
}

resource "aws_launch_configuration" "example" {
  image_id       = "${var.image_id}"
  instance_type = "${var.instance_type}"
  root_block_device {
    volume_type = "gp2"
    volume_size = 200
```

# The resources are in main.tf

```
# Foo Module for AWS

This is a Terraform Module to deploy
[Foo](http://www.example.com) on AWS, including:

* foo
* bar
* baz
```

**Documentation** is in README.md

# A more complicated module:

```
> tree complete-module
.
├── main.tf
├── outputs.tf
├── variables.tf
├── README.MD
├── modules
├── examples
└── test
```

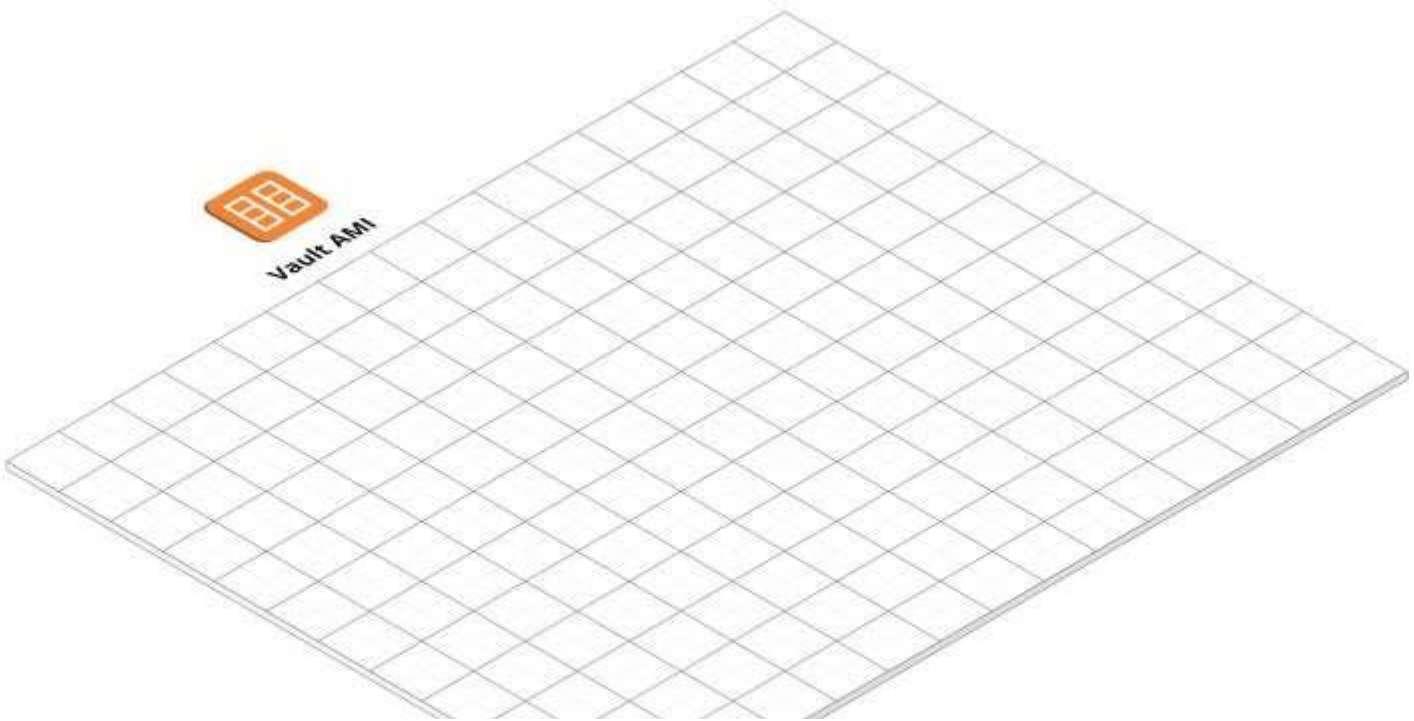
**We add three new folders:**
modules, examples, test

```
>tree  complete-module/modules
modules/
├── submodule-bar
│   ├── main.tf
│   ├── outputs.tf
│   └── variables.tf
└── submodule-foo
    ├── main.tf
    ├── outputs.tf
    └── variables.tf
```

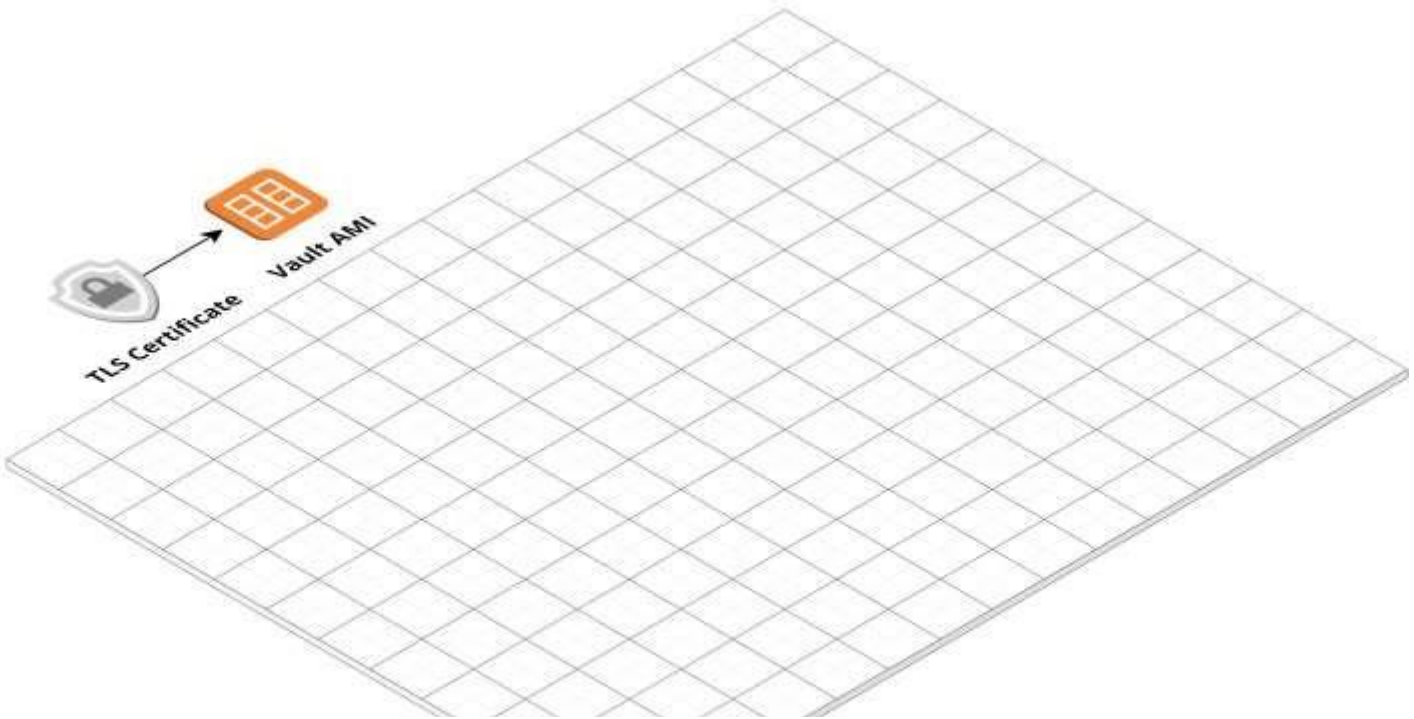
**The** modules **folder contains standalone “submodules”**

```
>tree  complete-module/modules
modules/
├── submodule-bar
│   ├── install-vault.sh
│   └── run-vault.sh
└── submodule-foo
    └── main.go
```

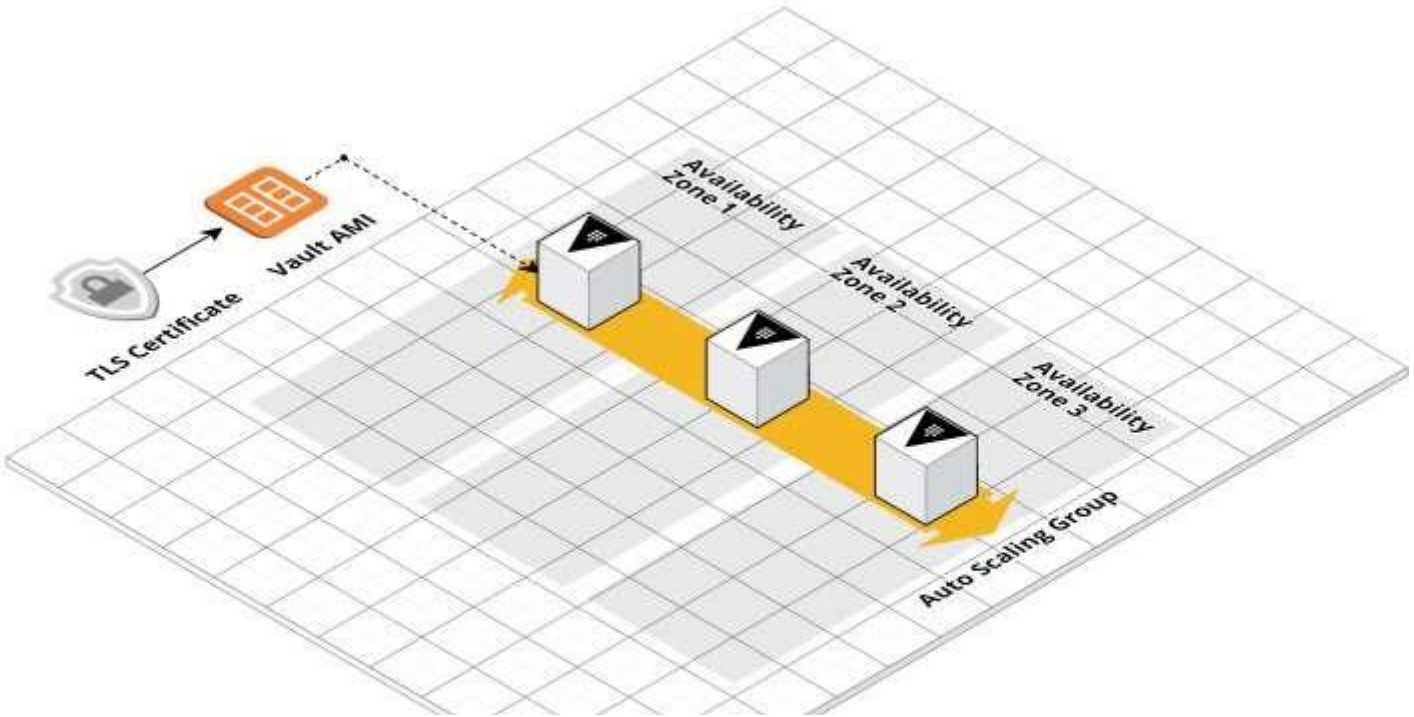
# Some of the submodules may not even be Terraform code



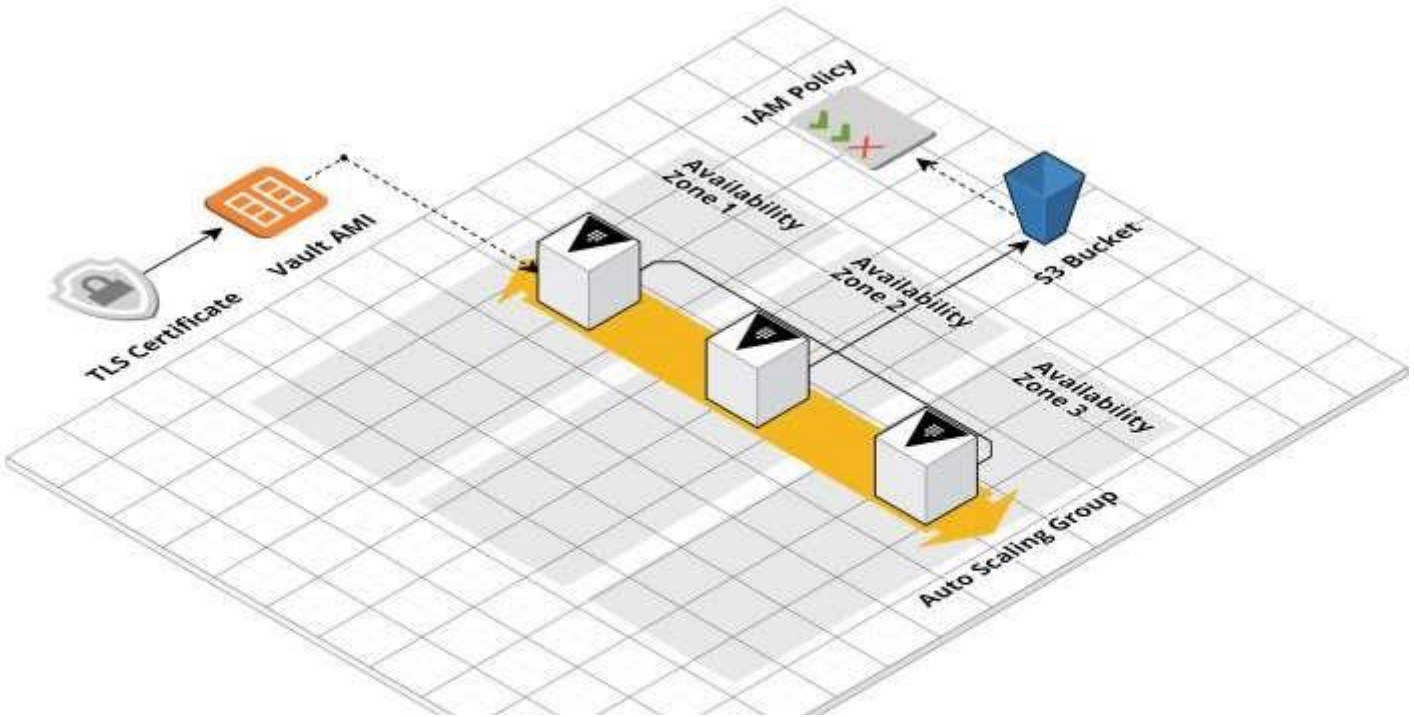

# For example, one submodule can be used to install Vault in an AMI



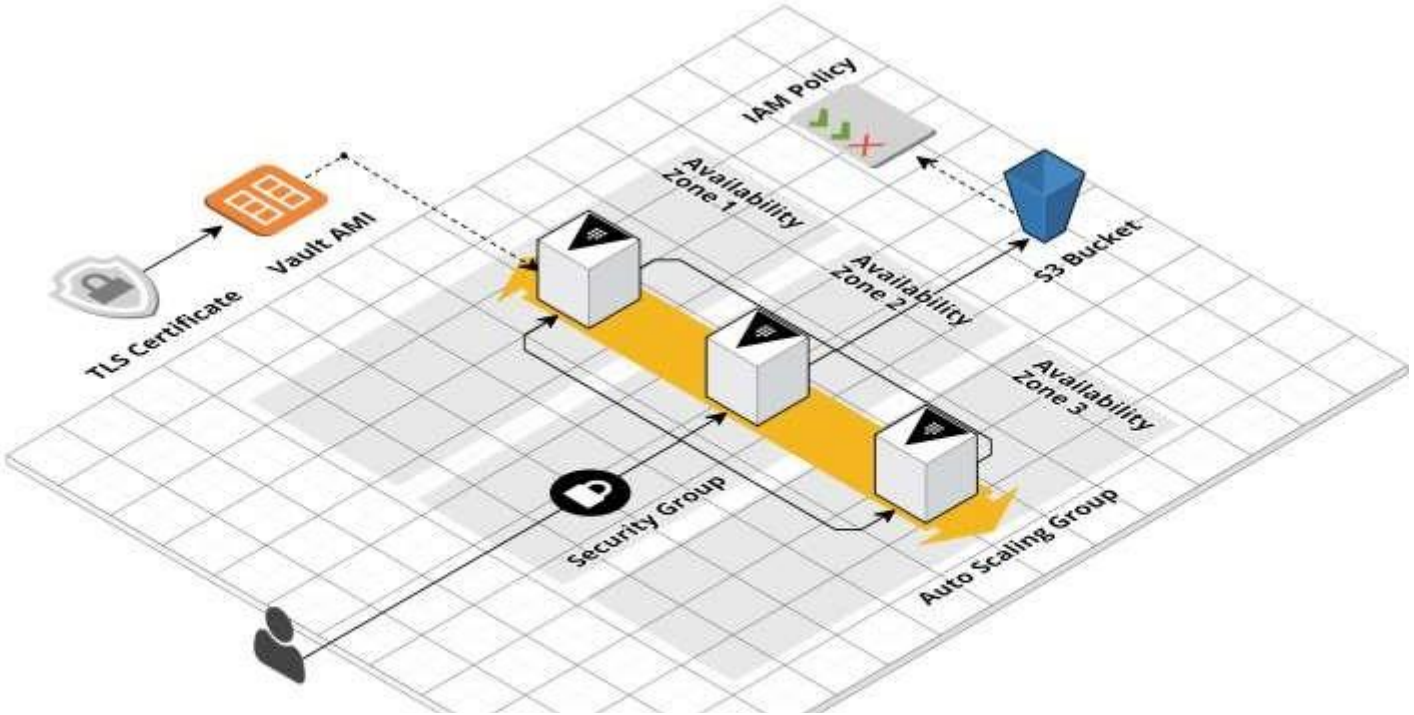

# Another to create self-signed TLS certificates



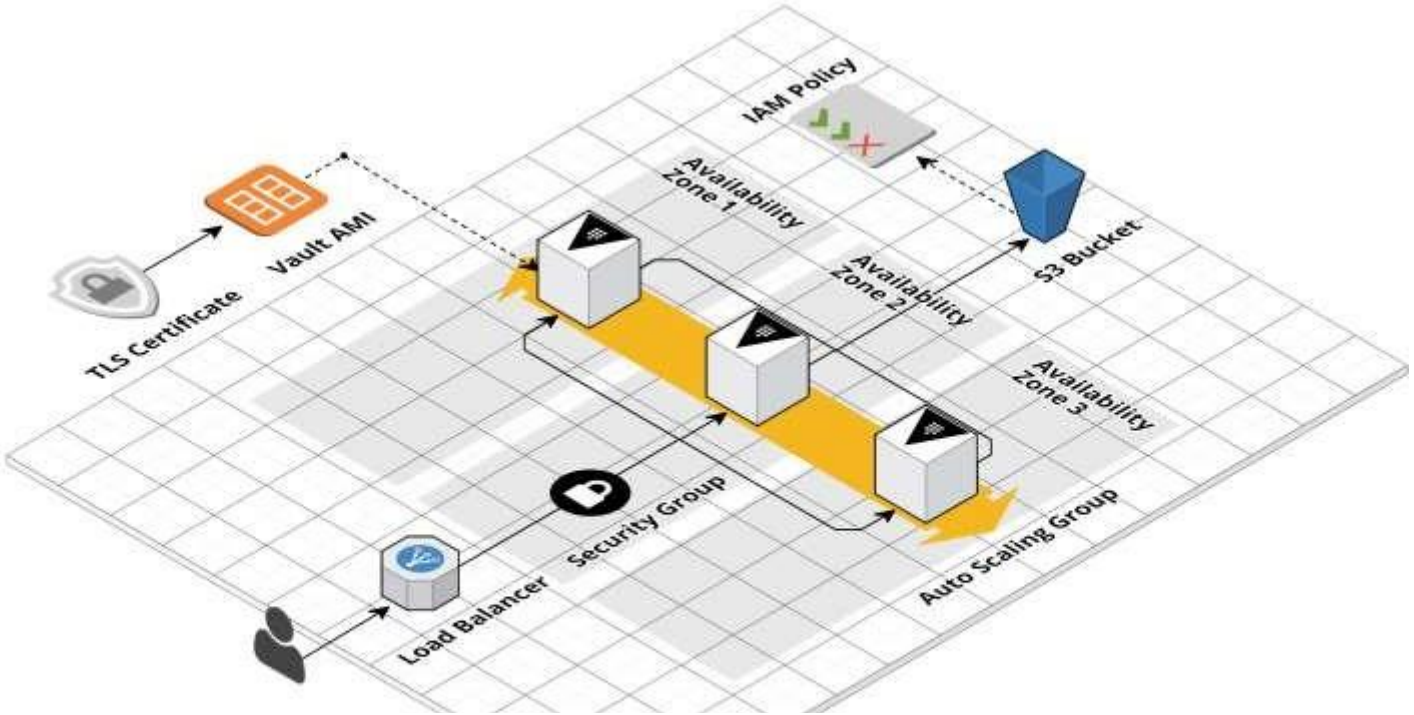

# Another to deploy the AMI across an Auto Scaling Group (ASG)



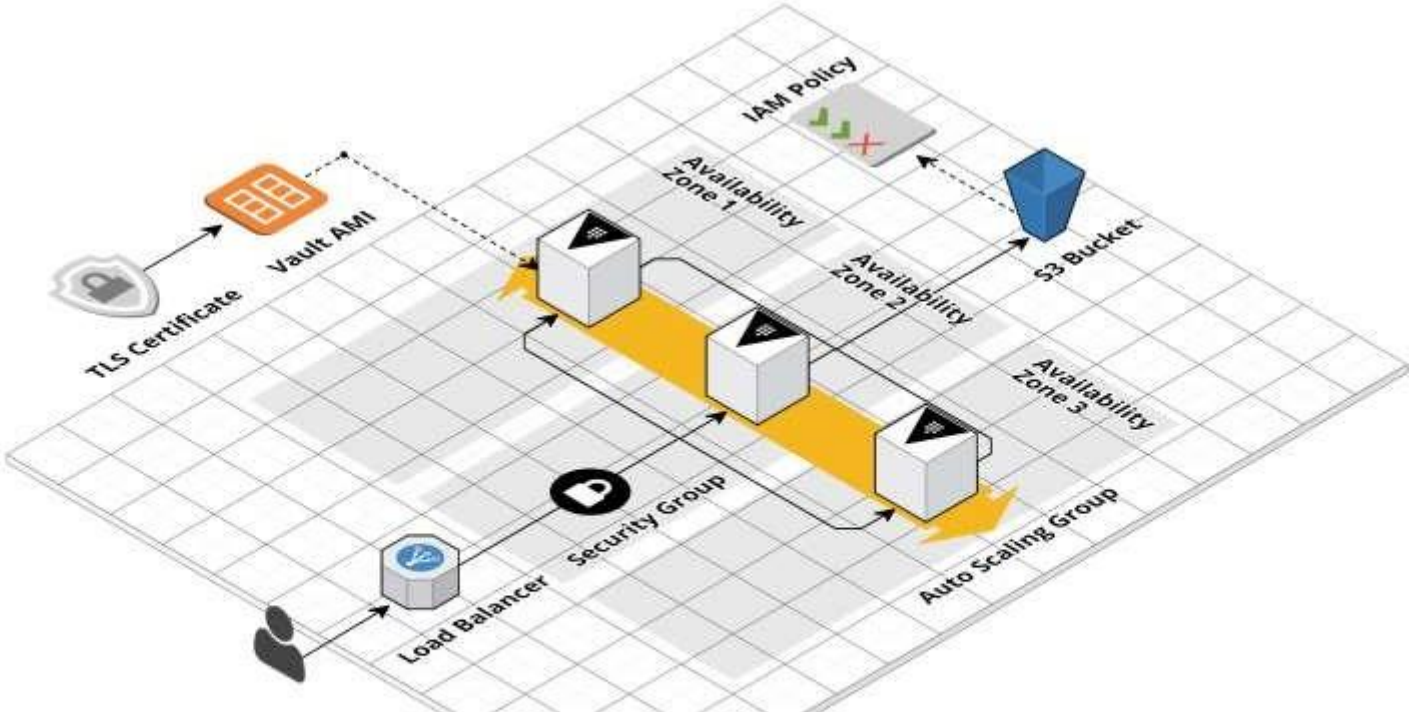

# Another to create an S3 bucket and IAM policies as a storage backend



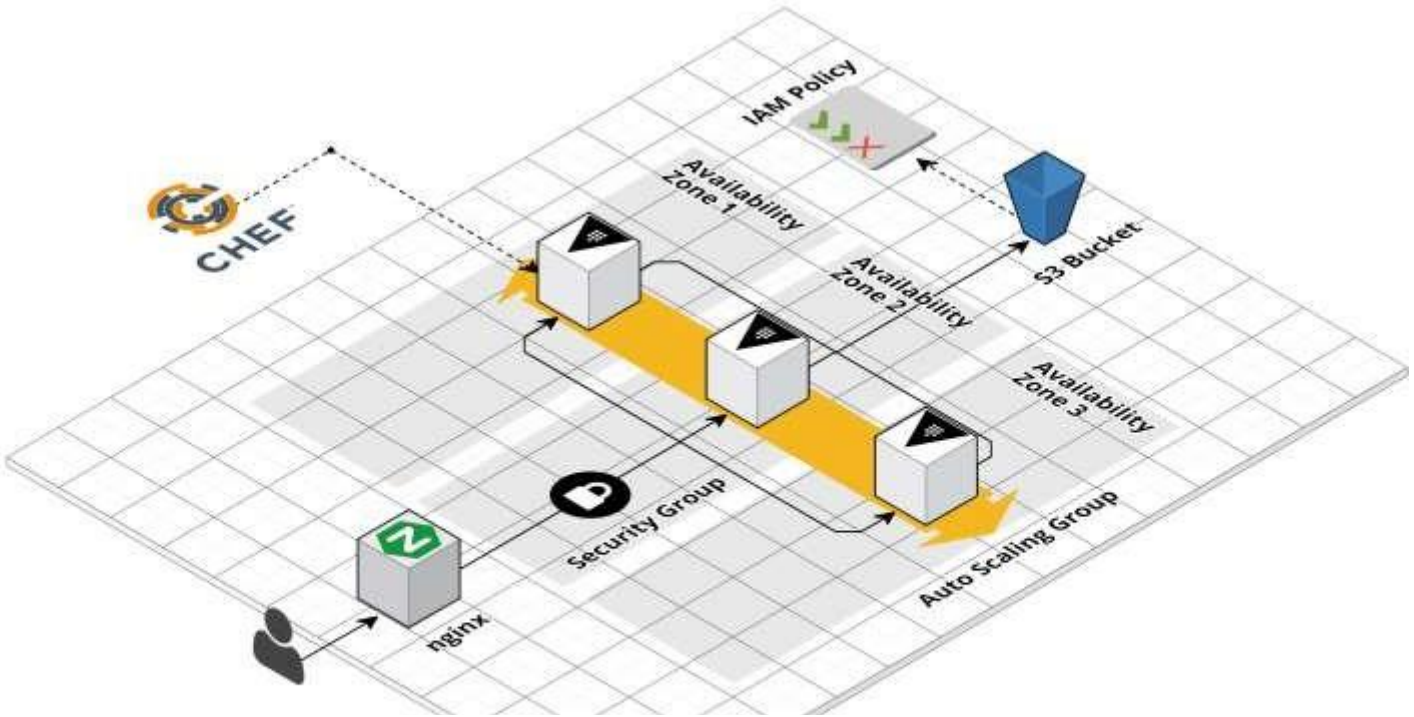

# Another to configure the Security Group settings




# And one more to deploy a load balancer (ELB)




# You can use all the submodules



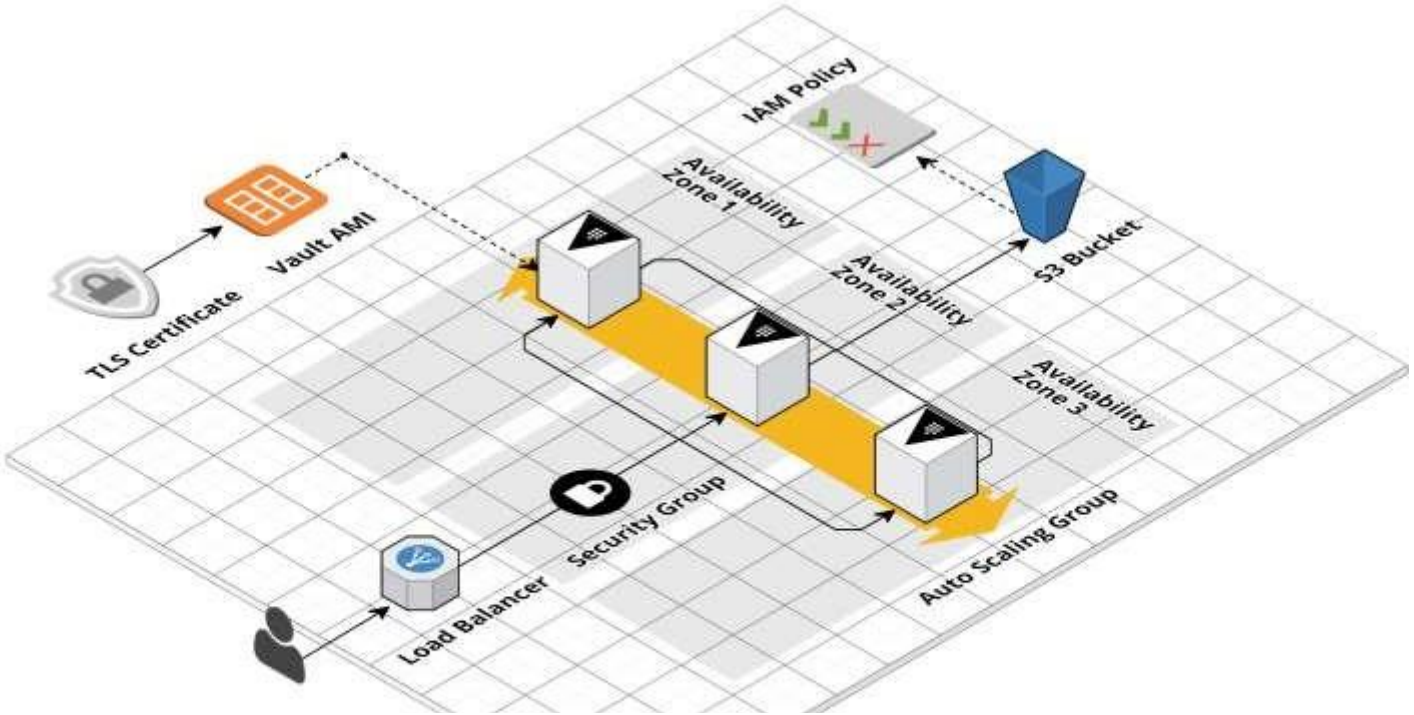

**Or pick the ones you want and swap in your own for the rest**

```
>tree  complete-module/examples
examples/
├── example-foo
│   ├── main.tf
│   ├── outputs.tf
│   └── variables.tf
└── example-bar
    ├── main.tf
    ├── outputs.tf
    └── variables.tf
```

**The** examples **folder shows how to use the submodules**

```
> tree complete-module
.
├── main.tf
├── outputs.tf
├── variables.tf
├── README.MD
├── modules
├── examples
└── test
```

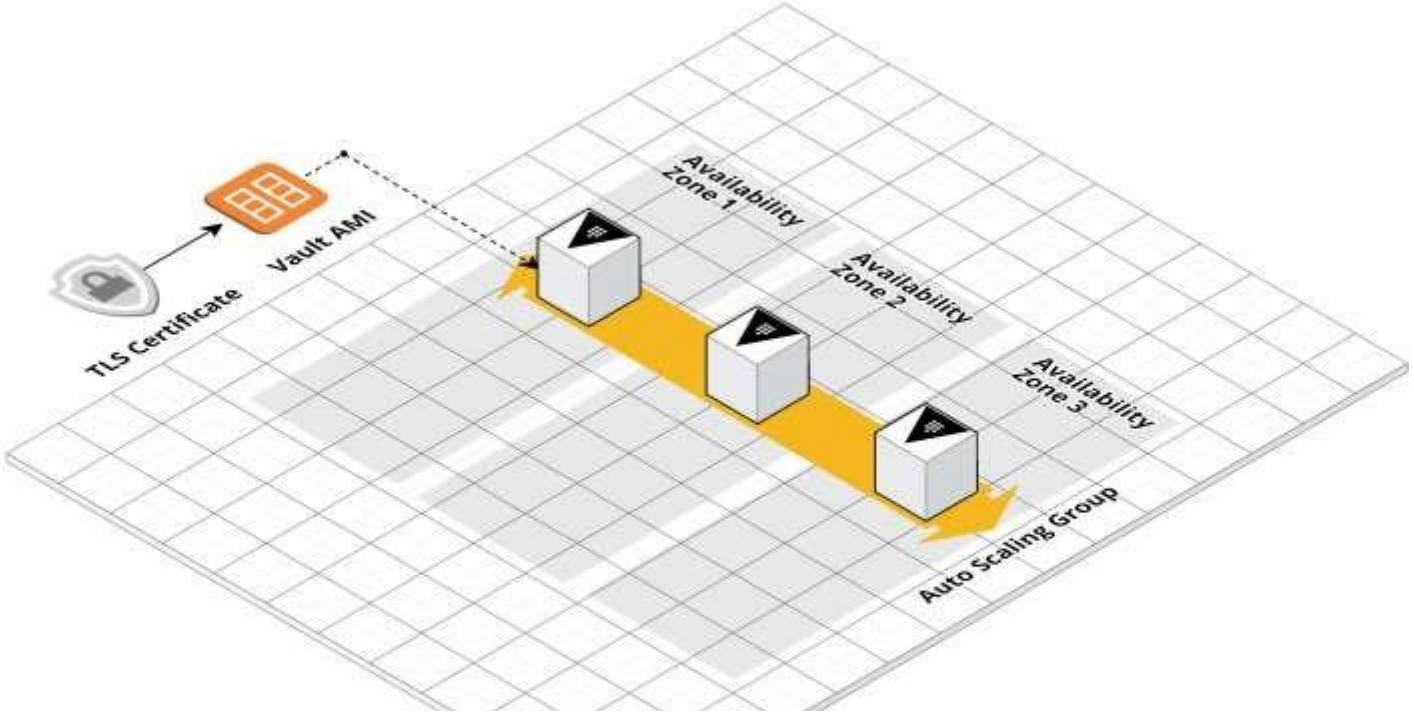
# Note: the code in the root is usually a “canonical” example



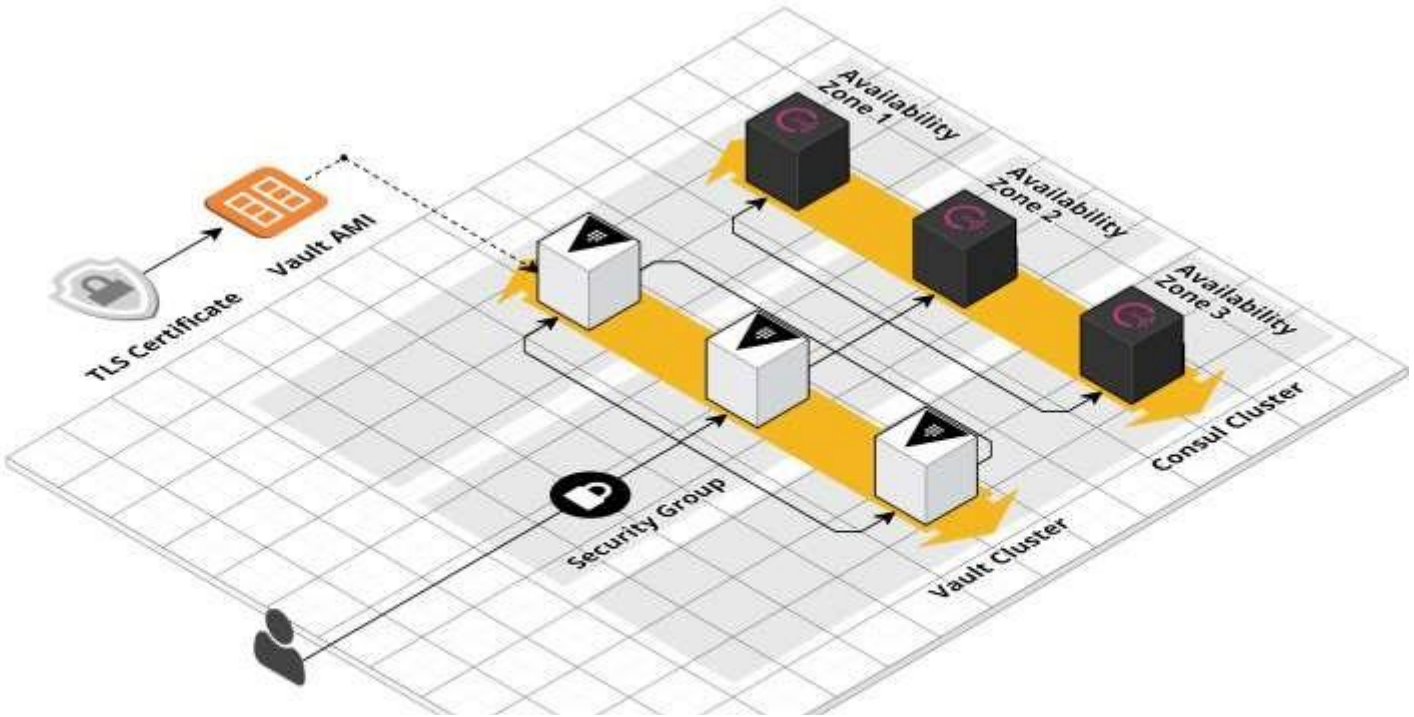

# It’s typically an opinionated way to use all the submodules together

```
>tree  complete-module/examples
examples/
├── example-foo
│   ├── main.tf
│   ├── outputs.tf
│   └── variables.tf
└── example-bar
    ├── main.tf
    ├── outputs.tf
    └── variables.tf
```

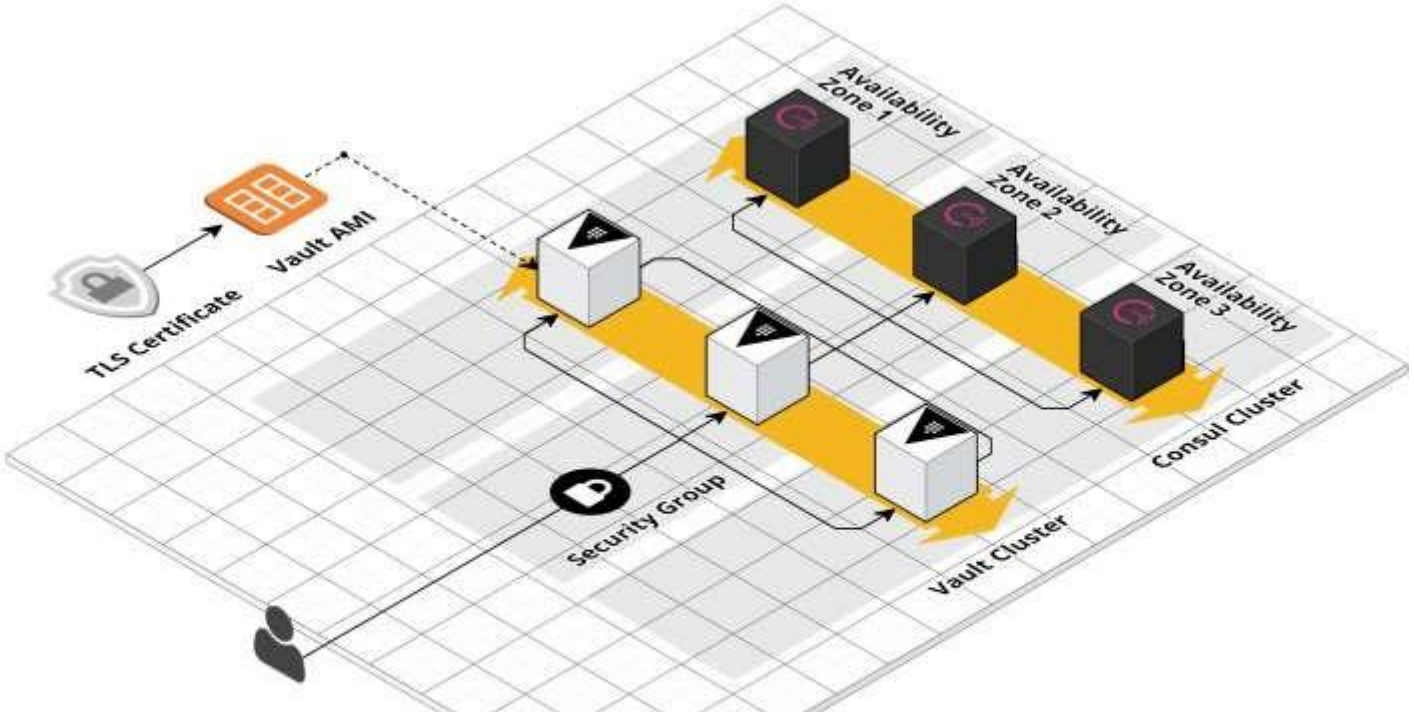
**The code in** examples **shows other possible permutations**




# E.g., How to use just one or two of the submodules together




# Or how to combine with other modules (e.g., Vault + Consul)




# This is like function composition!

```
>tree  complete-module/test
test/
├── example_foo_test.go
└── example_bar_test.go
```

# The `test` folder contains automated tests

```
>tree  complete-module/test
test/
├── example_foo_test.go
└── example_bar_test.go
```

# The tests are typically “integration tests”

```
func vaultTest(t *testing.T, options *terratest.Options) {
  tlsCert := generateSelfSignedTlsCert(t)
  defer cleanupSelfSignedTlsCert(t, tlsCert)

  amiId := buildVaultAmi(t)
  defer cleanupAmi(t, amiId)

  terratest.Apply(options)
  defer terratest.Destroy(options)

  assertCanInitializeAndUnsealVault(t, options)
}
```

# Example test case for Vault

```
func vaultTest(t *testing.T, options *terratest.Options) {
  tlsCert := generateSelfSignedTlsCert(t)
  defer cleanupSelfSignedTlsCert(t, tlsCert)

  amiId := buildVaultAmi(t)
  defer cleanupAmi(t, amiId)

  terratest.Apply(options)
  defer terratest.Destroy(options)

  assertCanInitializeAndUnsealVault(t, options)
}
```

# Create test-time resources

```
func vaultTest(t *testing.T, options *terratest.Options) {
  tlsCert := generateSelfSignedTlsCert(t)
  defer cleanupSelfSignedTlsCert(t, tlsCert)

  amiId := buildVaultAmi(t)
  defer cleanupAmi(t, amiId)

  terratest.Apply(options)
  defer terratest.Destroy(options)

  assertCanInitializeAndUnsealVault(t, options)
}
```

**Run** terraform apply

```
func vaultTest(t *testing.T, options *terratest.Options) {
  tlsCert := generateSelfSignedTlsCert(t)
  defer cleanupSelfSignedTlsCert(t, tlsCert)

  amiId := buildVaultAmi(t)
  defer cleanupAmi(t, amiId)

  terratest.Apply(options)
  defer terratest.Destroy(options)

  assertCanInitializeAndUnsealVault(t, options)
}
```

**Run terraform destroy at the end**

```
func vaultTest(t *testing.T, options *terratest.Options) {
  tlsCert := generateSelfSignedTlsCert(t)
  defer cleanupSelfSignedTlsCert(t, tlsCert)

  amiId := buildVaultAmi(t)
  defer cleanupAmi(t, amiId)

  terratest.Apply(options)
  defer terratest.Destroy(options)

  assertCanInitializeAndUnsealVault(t, options)
}
```

# Check the Vault cluster works!

# Using a module:

```
module "service_foo" {
  source    = "./minimal-module"

  name      = "Foo"
  image_id  = "ami-123asd1"
  port      = 8080
}
```

**For simple Modules and learning, deploy the root**

```
module "submodule_foo" {
  source    = "./complete-module/modules/submodule-foo"
  param_foo = "foo"
  param_bar = 8080
}
module "submodule_bar" {
  source    = "./complete-module/modules/submodule-bar"
  param_foo = "abcdef"
  param_bar = 9091
}
```

**For more complicated use-cases, use the submodules**

```
module "service_foo" {
  source    = "./minimal-module"
  name      = "Foo"
  image_id  = "ami-123asd1"
  port      = 8080
}
```

**Abstraction:** simple Module API for complicated infrastructure

```
module "service_foo" {
  source    = "./minimal-module"
  name      = "Foo"
  image_id  = "ami-123asd1"
  port      = 8080
}
module "service_bar" {
  source    = "./minimal-module"
  name      = "Bar"
  image_id  = "ami-abcd1234"
}
```

**Re-use: create a Module once, deploy it many times**

# Versioning:

```
module "service_foo" {
  source    = "./foo"

  name      = "Foo"
  image_id  = "ami-123asd1"
  port      = 8080
}
```

**You can set `source` to point to modules at local file paths**

```
module "service_foo" {
  source    = "hashicorp/vault/aws"

  name      = "Foo"
  image_id = "ami-123asd1"
  port      = 8080
}
```

# Alternatively, you can use Terraform registry URLs

```
module "service_foo" {
  source    = "git::git@github.com:foo/bar.git"

  name      = "Foo"
  image_id = "ami-123asd1"
  port      = 8080
}
```

# Or arbitrary Git URLs

```
module "service_foo" {
  source    = "git::git@github.com:foo/bar.git?ref=v1.0.0"

  name      = "Foo"
  image_id = "ami-123asd1"
  port      = 8080
}
```

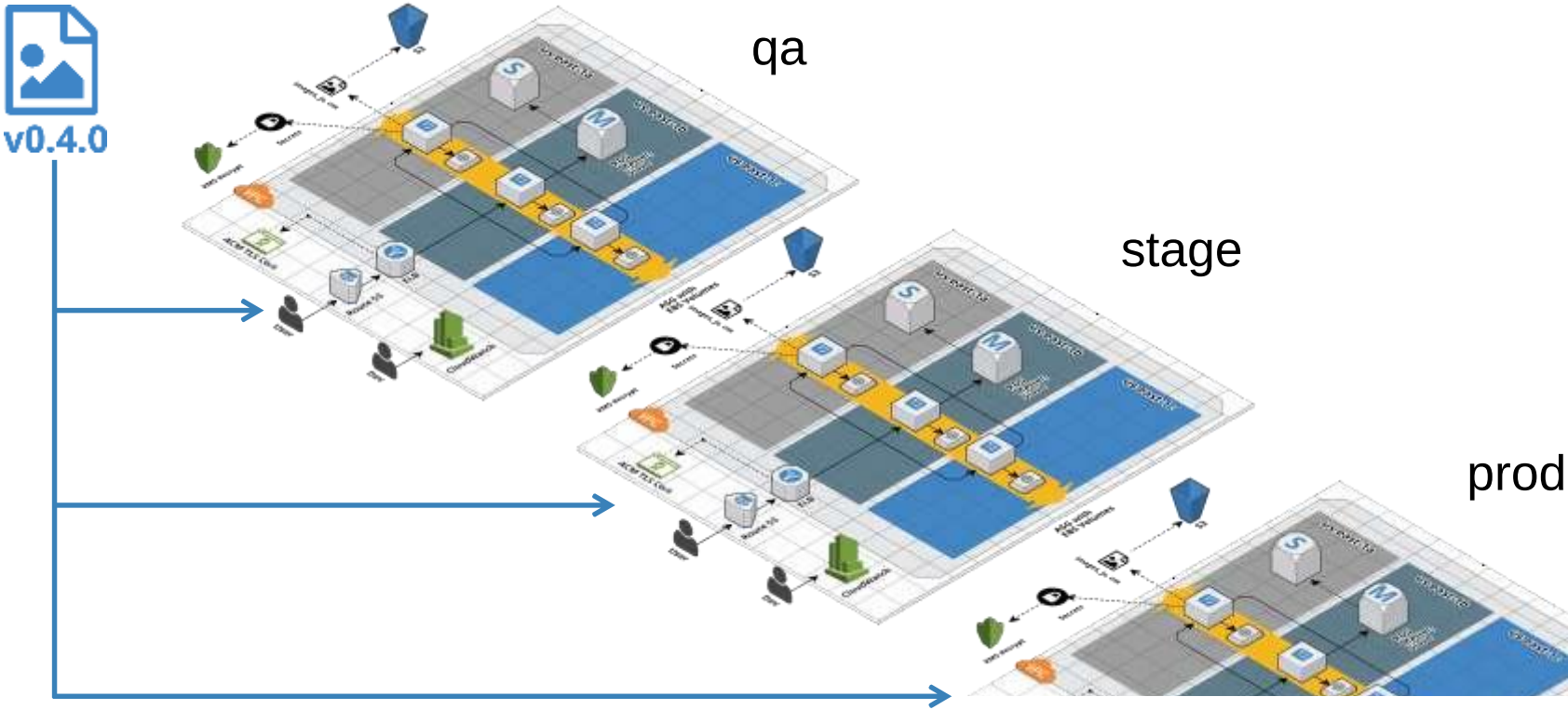
**You can even link to a specific Git tag (recommended!)**

```
module "service_foo" {
  source   = "git::git@github.com:foo/bar.git?ref=v1.0.0"

  name     = "Foo"
  image_id = "ami-123asd1"
  port     = 8080
}
```

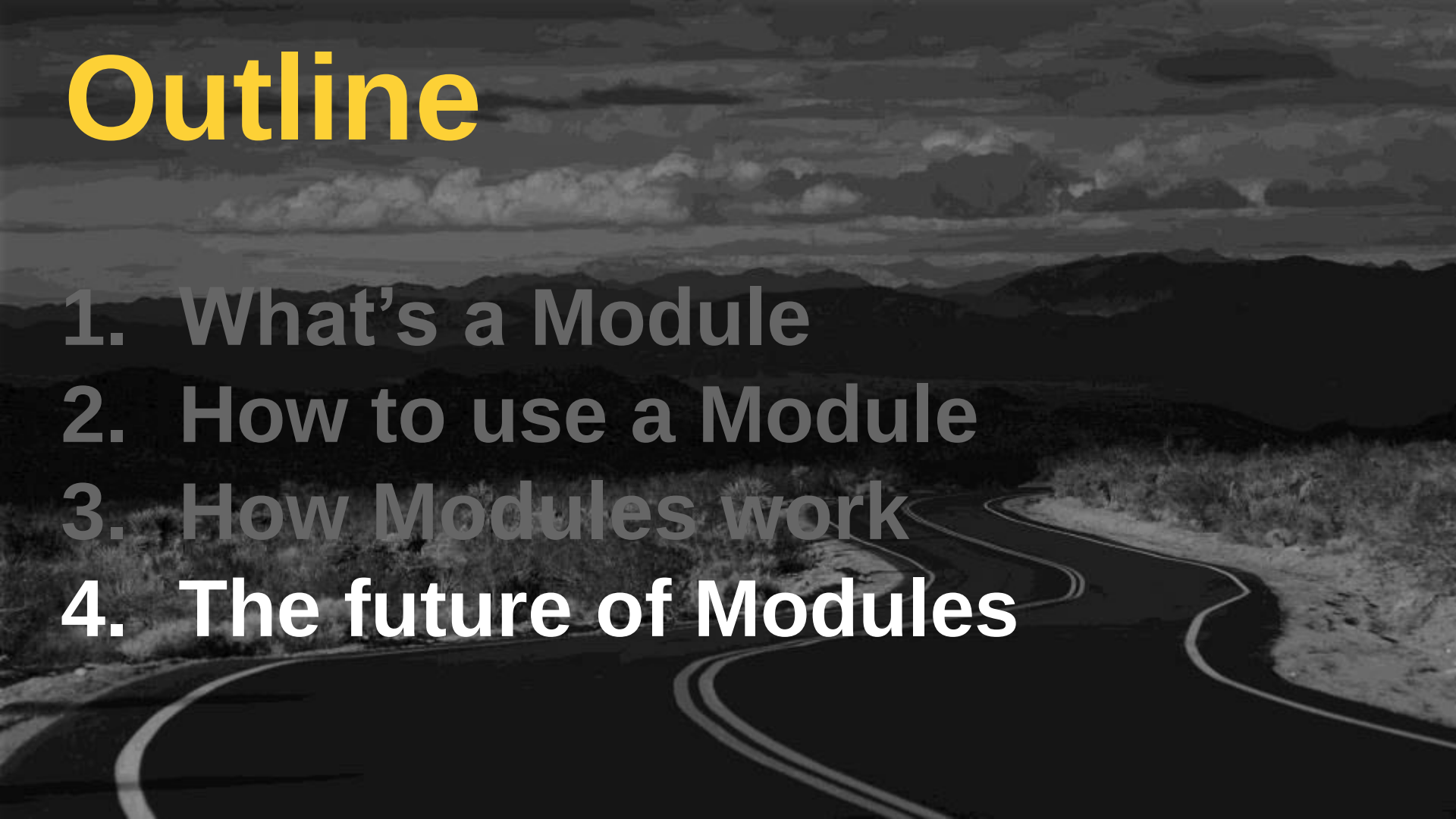
# Modules use semantic versioning

```
module "service_foo" {
  source    = "git::git@github.com:foo/bar.git?ref=v2.0.0"

  name      = "Foo"
  image_id = "ami-123asd1"
  port      = 8080
}
```

**So upgrading infrastructure is just a version number bump**




# Promote immutable, versioned infrastructure across environments

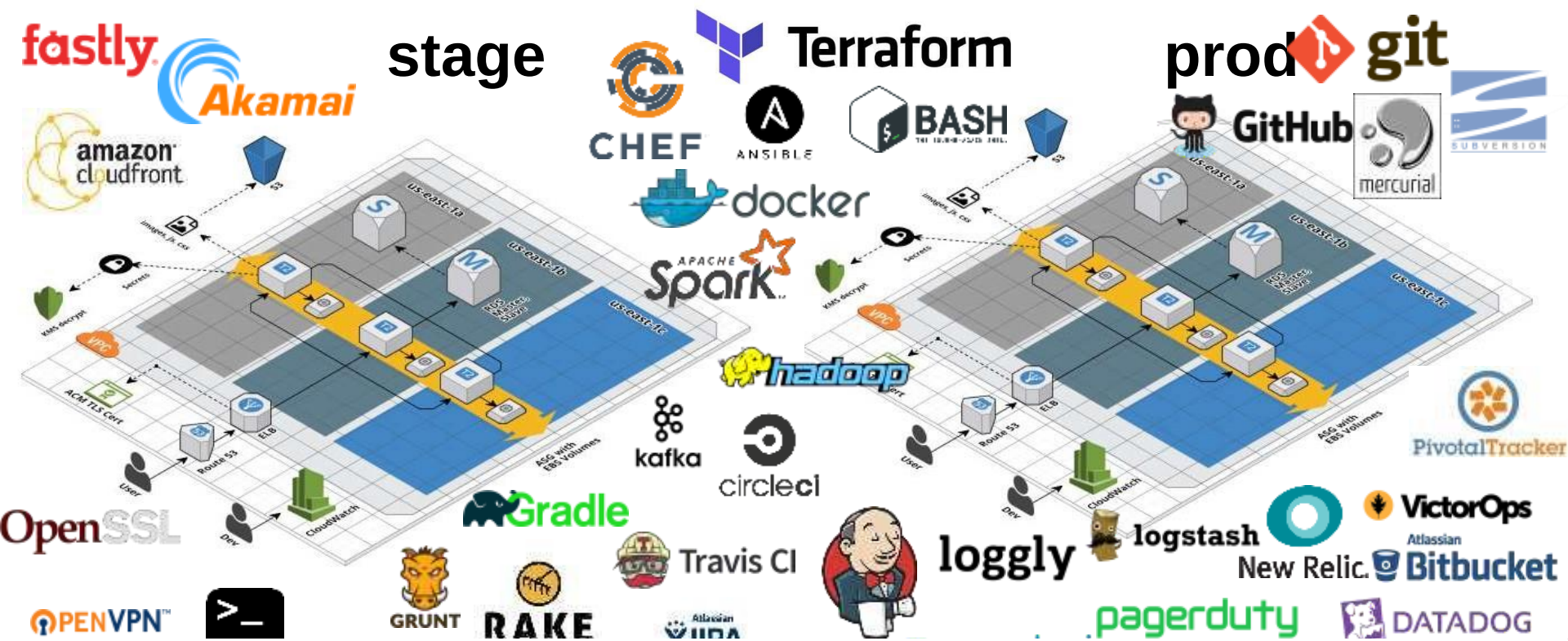


# Outline

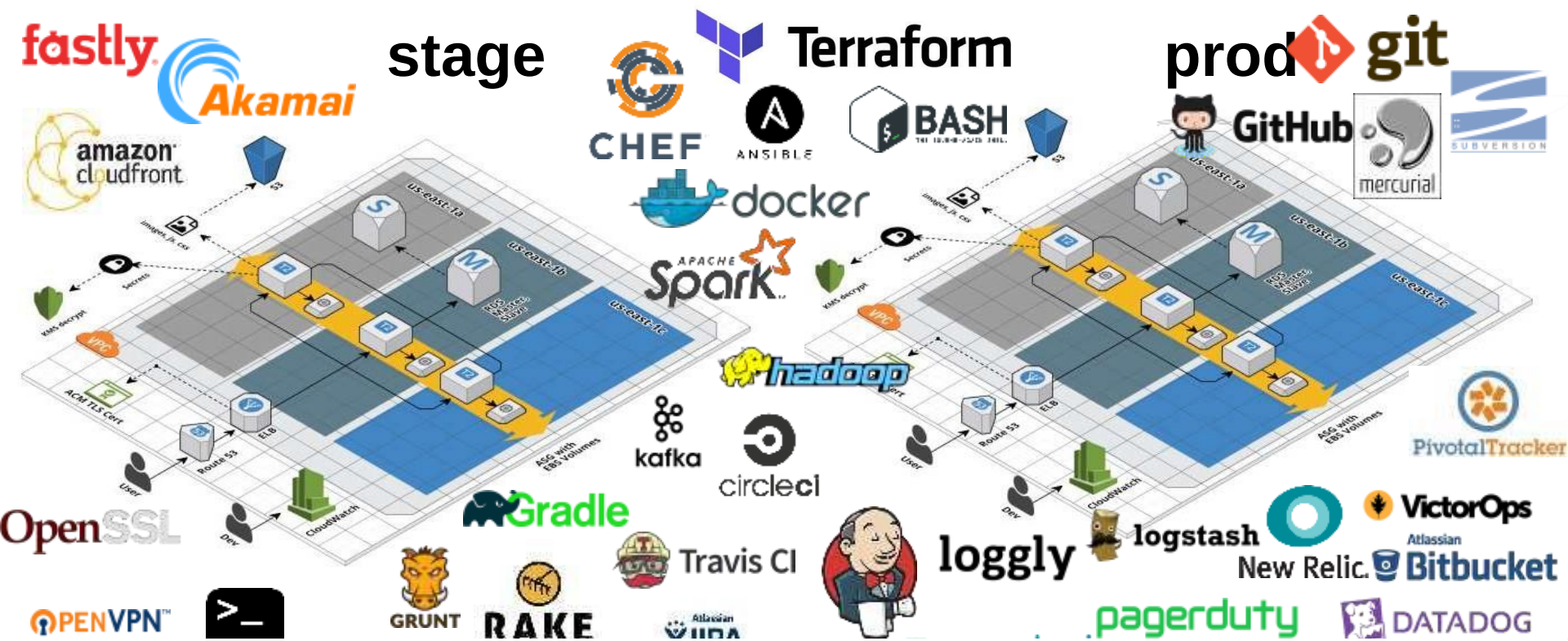
1. What's a Module
2. How to use a Module
3. How Modules work
4. The future of Modules

*u are not special.  Your*
*astructure is  not a beautiful*
*d  unique snowflake. You*
*ve the same tech debt  as*
*eryone else.”*

our sysadmin,  probably




# You need this




# And so does everyone else

# Stop reinventing the wheel

# Start building on top of battle-tested code

# Start building on top of commercially-supported code

# Start building on top of code

# Advantages of code

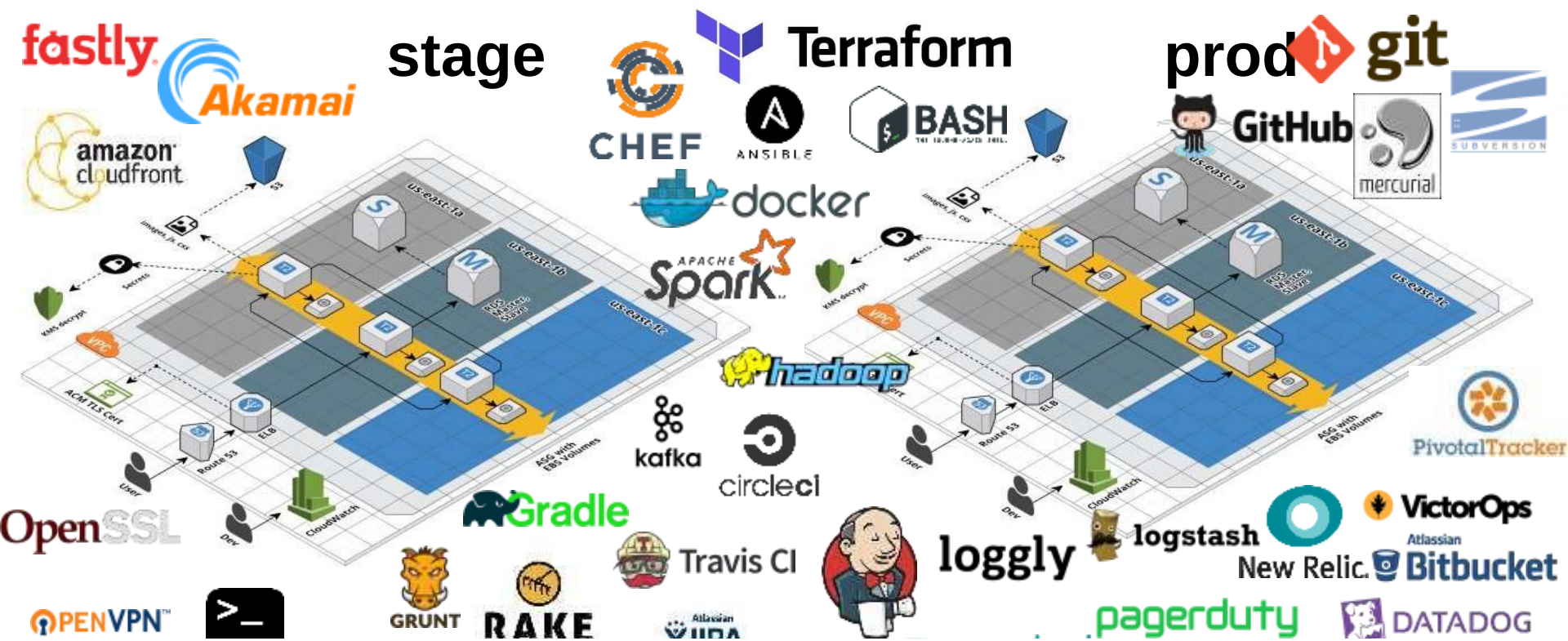
1. Reuse
2. Compose
3. Configure
4. Customize
5. Debug
6. Test
7. Version
8. Document

gruntwork.io

**At Gruntwork, we've been building Modules for years**

gruntwork.io

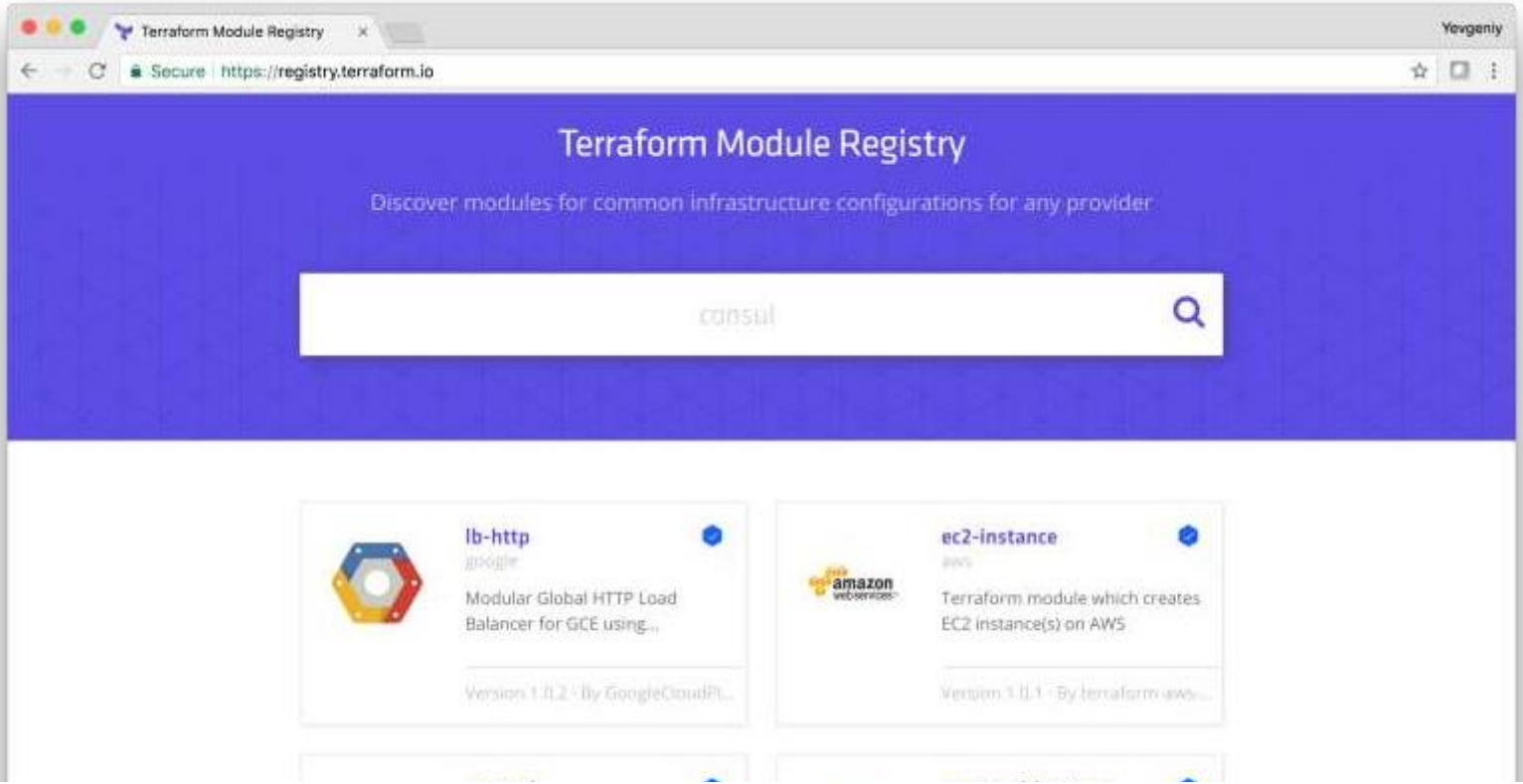
**Many companies, all running on the same infrastructure code**



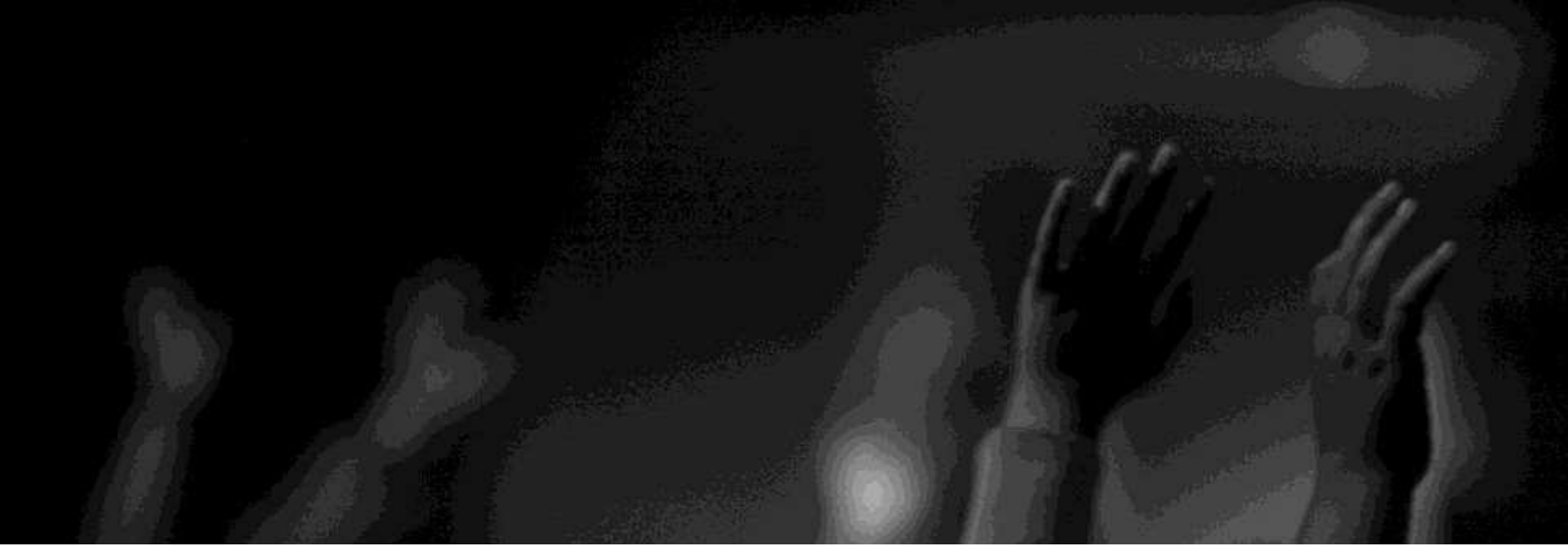

# Modules allow us to turn this…

```
> terraform init <..>

> terraform apply
```

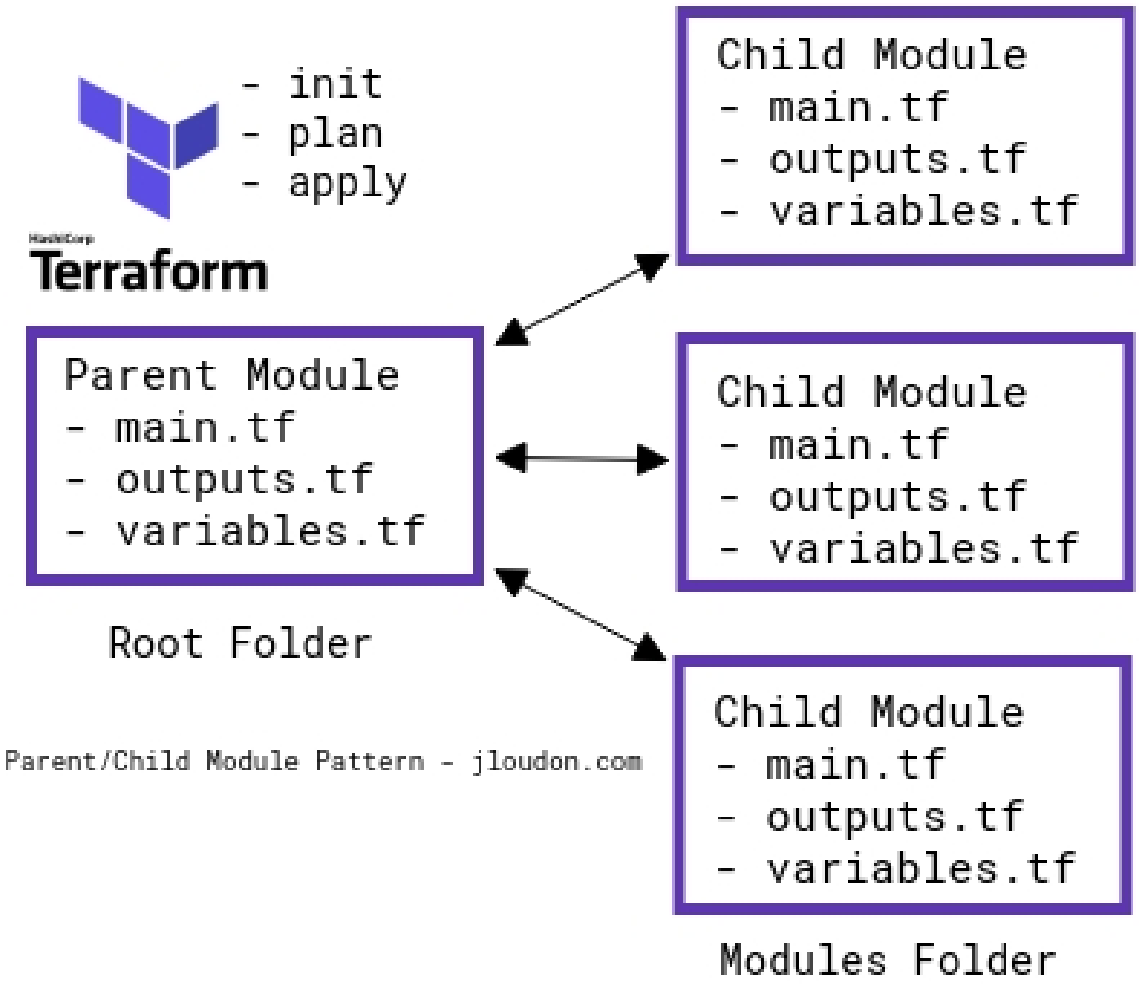
# … into this

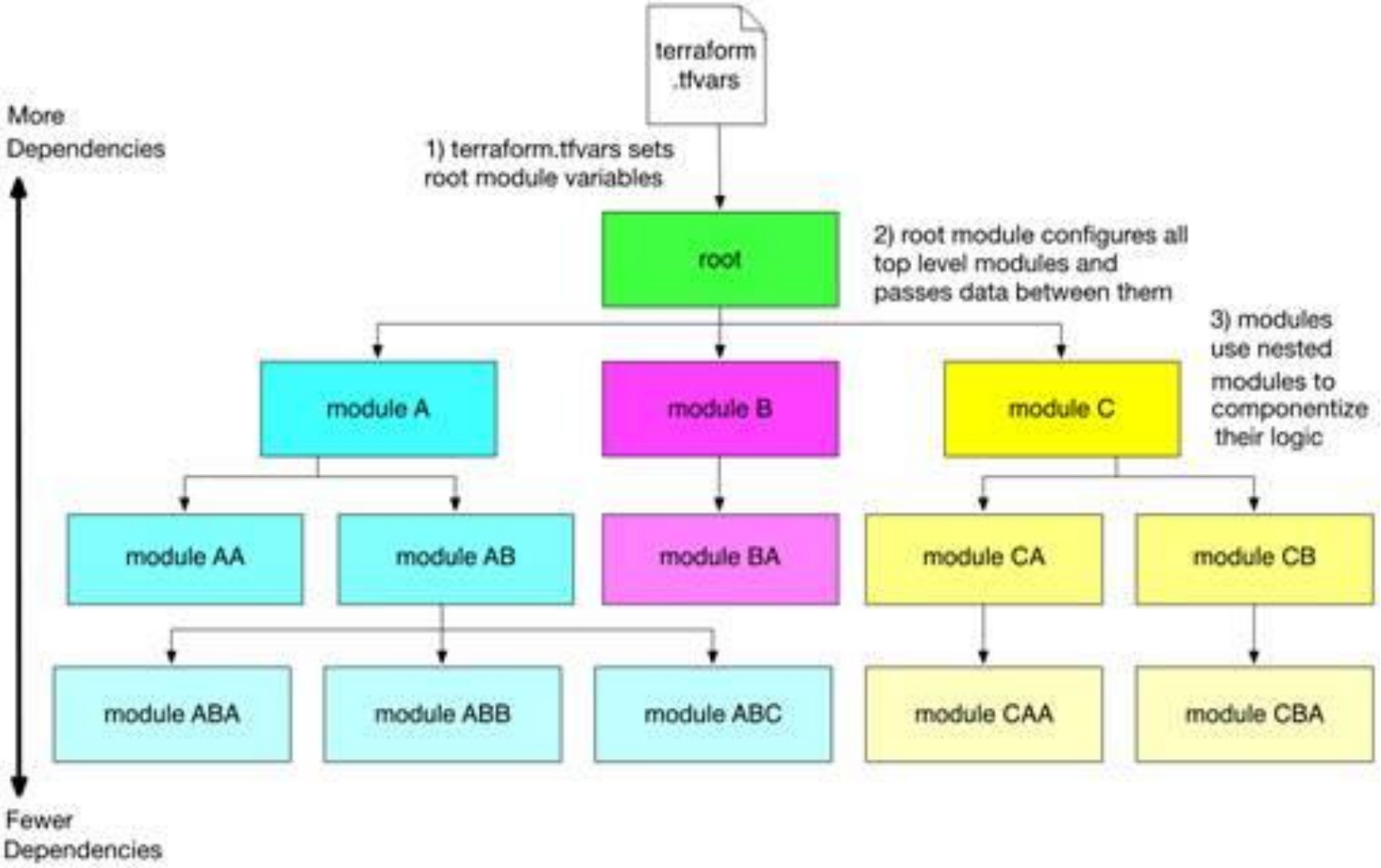


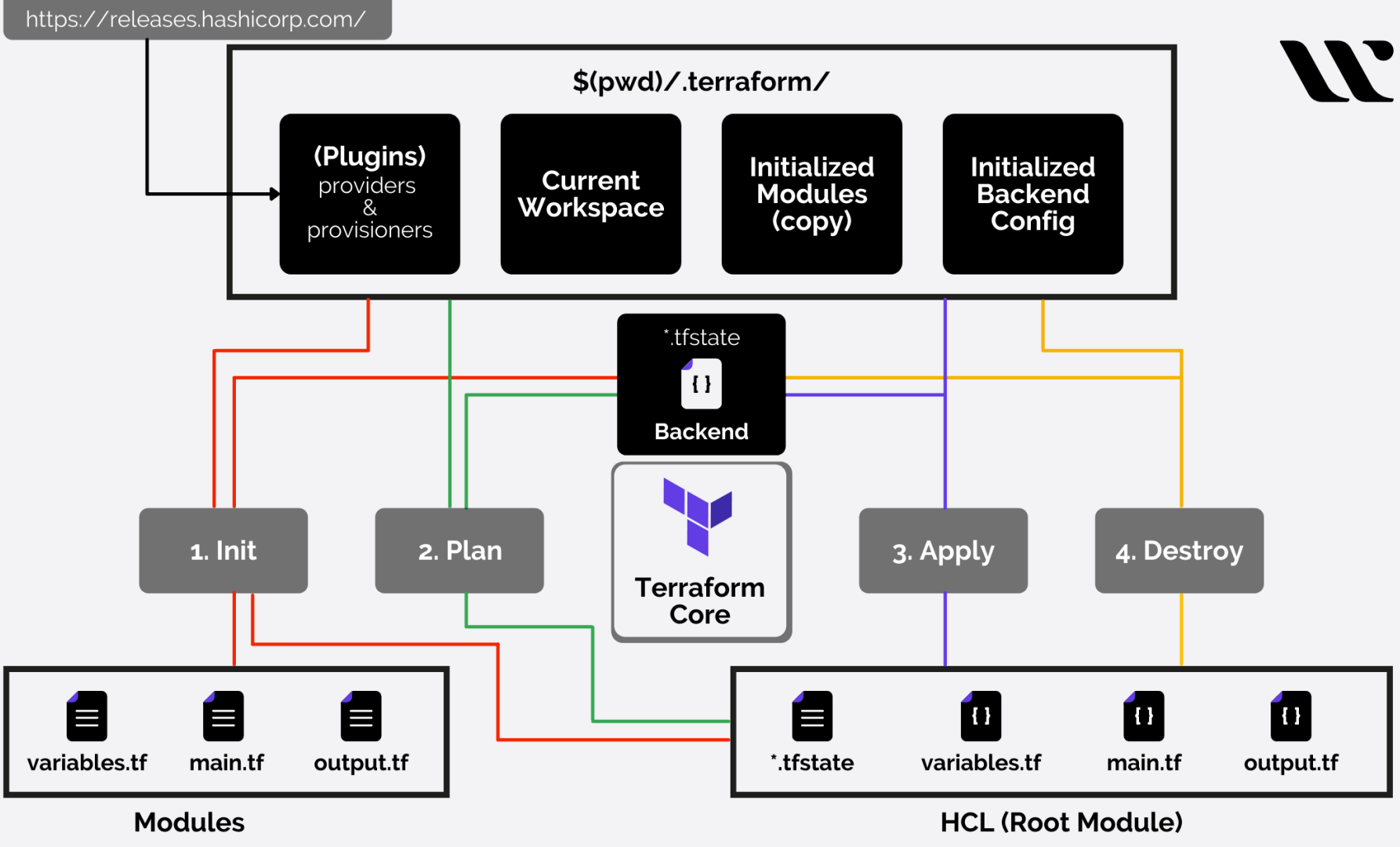

# With some help from this



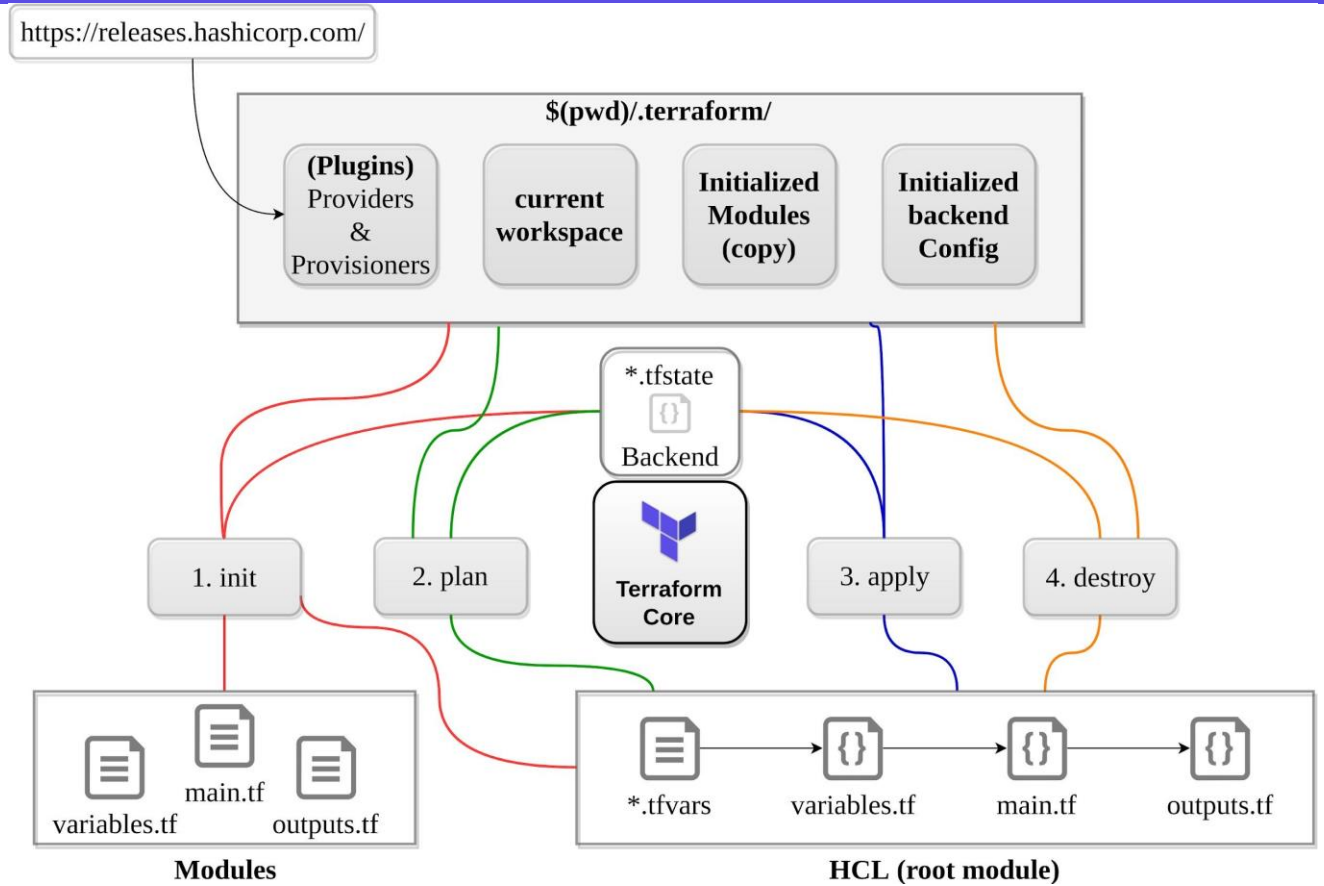
# Questions?


- init
- plan
- apply
Terraform
Parent Module
- main.tf
- outputs.tf
- variables.tf
Root Folder
Child Module
- main.tf
- outputs.tf
- variables.tf
Child Module
- main.tf
- outputs.tf
- variables.tf
Child Module
- main.tf
- outputs.tf
- variables.tf
Modules Folder
Parent/Child Module Pattern - jloudon.com


terraform
.tfvars
More
Dependencies
1) terraform.tfvars sets
root module variables
root
2) root module configures all
top level modules and
passes data between them
3) modules
use nested
modules to
componentize
their logic
module A
module B
module C
module AA
module AB
module BA
module CA
module CB
module ABA
module ABB
module ABC
module CAA
module CBA
Fewer
Dependencies

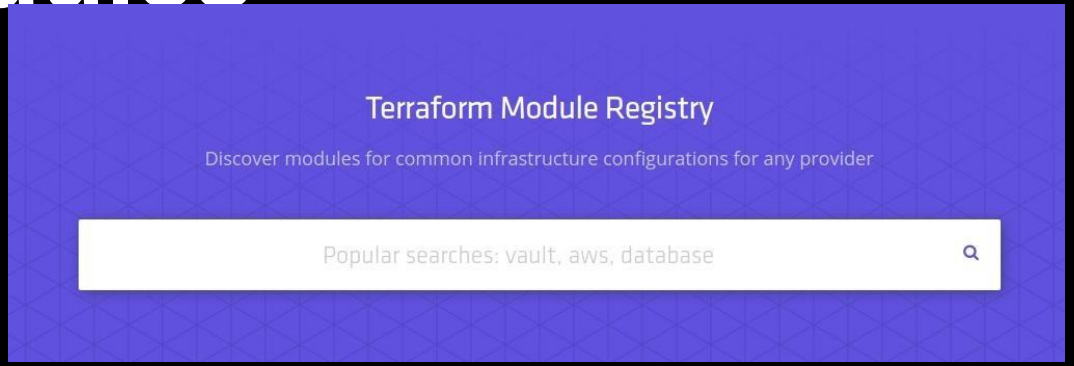

https://releases.hashicorp.com/
$(pwd)/.terraform/
(Plugins)
providers
&
provisioners
Current
Workspace
Initialized
Modules
(copy)
Initialized
Backend
Config
*.tfstate
{ }
Backend
1. Init
2. Plan
Terraform
Core
3. Apply
4. Destroy
variables.tf
main.tf
output.tf
Modules
*.tfstate
variables.tf
main.tf
output.tf
HCL (Root Module)

# Simple workflow



# Workflow: Modules

- Code reuse
- Apply versioning
- Use version constraints
- Store code remotely
- Easier testing
- Encapsulation
- Use and contribute to Module Registry



# Workflow: Modules

What is the good terraform module?

- Clean and flexible code
- Well presented default values
- Covered with tests
- Examples
- Documentation
- Changelog
- Secure

Do not overload modules with features:
Terraform does cloning everything

https://www.endava.com/en/blog/Engineering/2019/11-Things-I-wish-I-knew-before-working-with-Terraform-I

# Terraform Module

```
variable “name” {}

resource “aws_s3_bucket” “bucket”{
  name = “${var.name}”
  […]
}

output “bucket_id” {
  value = “${aws_s3_bucket.bucket.id}”
}
```

# Terraform Module

```
#Create module bucket
module “bucket” {
  name = “MahBucket”
  source = “.\\Modules\\s3”
}

#Use MahBucket
resource “aws_s3_bucket_object” {
  bucket = “${module.bucket.bucket_id}”
  key = “/walrus/bucket.txt”
  source = “./mahbucket.txt”
}
```

# Use a Module

## Use a module from git

```
module "module-example" {
    source = "github.com/wardviaene/terraform-module-example"
}
```

## Use a module from a local folder

```
module "module-example" {
    source = "./module-example"
}
```

# Module Parameter

Pass arguments to the module

```
module "module-example" {
    source = "./module-example"
    region = "us-west-1"
    ip-range = "10.0.0.0/8"
    cluster-size = "3"
}
```