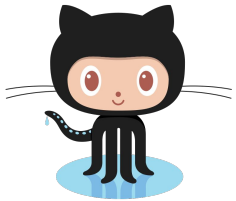
Migrating HealthCare.gov to Terraform: Lessons Learned

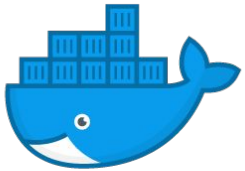
Christian Monaghan
@monaghan_a_gram
Cofounder, Nava PBC

What is Terraform?

A tool for **building**, **changing**, and **versioning** infrastructure

Manage cloud providers





Infrastructure as Code

- Declarative syntax
- Source control
- Variable support

```
provider "aws" {
  access_key = "super_secret_key"
  secret_key = "another_secret_key"
  region     = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-2757f631"
  instance_type = "t2.micro"
}

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

Execution plans

- Developer reviews plan before proceeding

```
$ terraform apply

+ aws_eip.ip
    domain:             "<computed>"
    instance:           "${aws_instance.example.id}"
    network_interface:  "<computed>"
    private_ip:         "<computed>"
    public_ip:          "<computed>"
    ...

+ aws_instance.example
    ami:                          "ami-b374d5a5"
    availability_zone:            "<computed>"
    instance_state:               "<computed>"
    instance_type:                "t2.micro"
    key_name:                     "<computed>"
    private_dns:                  "<computed>"
    private_ip:                   "<computed>"
    public_dns:                   "<computed>"
    public_ip:                    "<computed>"
    source_dest_check:            "true"
    ...
```

Resource graph

- Resources created in dependency order

```
# ...
aws_instance.example: Creating...
  ami:                          "" => "ami-b374d5a5"
  instance_type:                "" => "t2.micro"
  [..]
aws_instance.example: Still creating... (10s elapsed)
aws_instance.example: Creation complete
aws_eip.ip: Creating...
  allocation_id:      "" => "<computed>"
  association_id:     "" => "<computed>"
  domain:             "" => "<computed>"
  instance:           "" => "i-f3d77d69"
  network_interface:  "" => "<computed>"
  private_ip:         "" => "<computed>"
  public_ip:          "" => "<computed>"
aws_eip.ip: Creation complete

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

Resource graph

- Resources created in dependency order

```
# ...
aws_instance.example: Creating...
  ami:                      "" => "ami-b374d5a5"
  instance_type:            "" => "t2.micro"
  [..]
aws_instance.example: Still creating... (10s elapsed)
aws_instance.example: Creation complete
aws_eip.ip: Creating...
  allocation_id:      "" => "<computed>"
  association_id:     "" => "<computed>"
  domain:             "" => "<computed>"
  instance:           "" => "i-f3d77d69"
  network_interface:  "" => "<computed>"
  private_ip:         "" => "<computed>"
  public_ip:          "" => "<computed>"
aws_eip.ip: Creation complete

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

Our project history

AWS Cloudformation

JSON interface

3,000+ lines for 1 Virtual Private Cloud (VPC)

Managing dozens of VPCs

Custom tooling to interact with Cloudformation

Challenges we faced with our existing tooling

Maintaining custom code :(

- Complex
- Not unit tested
- Limited documentation, quickly out of date
- Increasing bloat
- Hard to understand
- Hard to debug

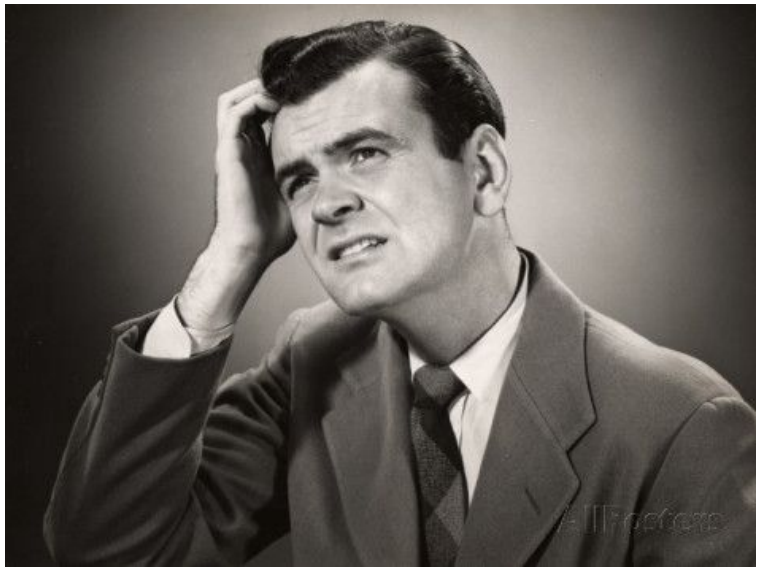
Unable to incorporate manual changes

Past examples:

- Horizontally scale NATs (Network Address Translation)
- Adding a temporary second Elastic Load Balancer
- Scaling down from 3 availability zones to 1 availability zone
- Swap in new Elastic IPs

Uncertain client demands

- Must build atop partially provisioned vpc infrastructure
- Client frequently requesting custom architecture changes
- Client might make manual changes that would be unrecoverable in Cloudformation



Proliferating use cases

- Load testing resources
- Continuous Integration clusters
- Custom monitoring
- Graphite/Graphana
- Nessus scanning clusters

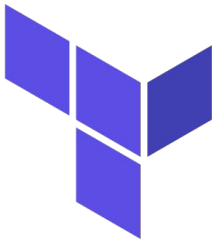
We were trying to shoehorn all these new use cases into our existing tooling

Engineering goal

Manage all infrastructure with a single tool that is **flexible**, **extensible**, **fast**, and **well-supported**

Choosing the right tool

Tools we considered



HashiCorp
Terraform

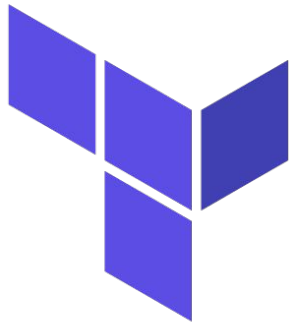
CHEF

Chef, Puppet, Ansible, SaltStack

- These are **configuration management** tools
- Install and manage software on **existing** machines

Why we chose Terraform

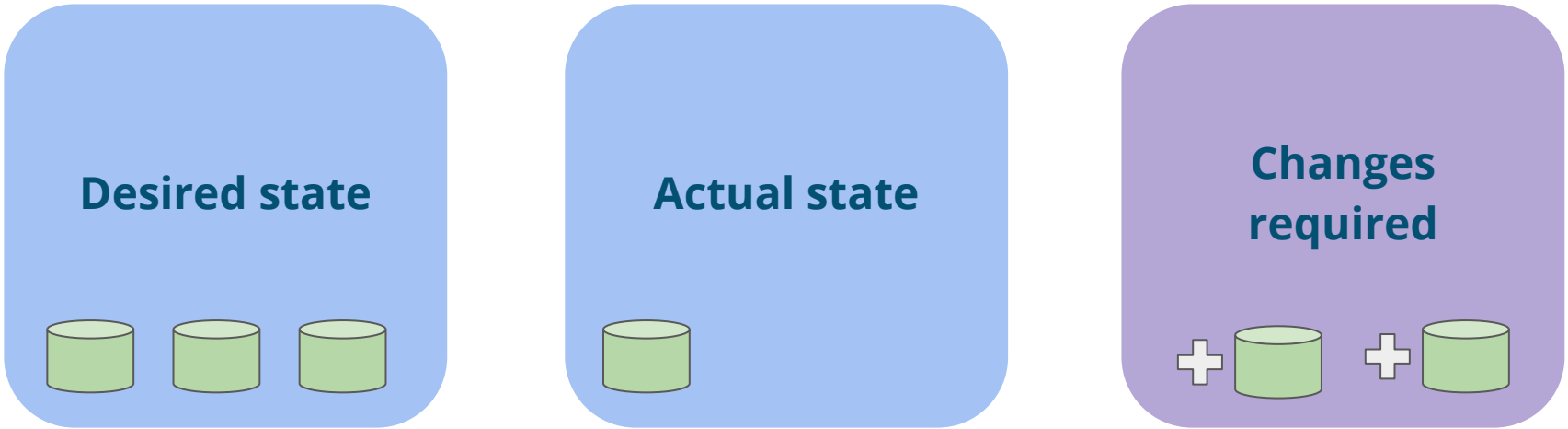
- Incorporate manual changes
- Declarative syntax, easy to read, understand, extend
- Supports multiple providers
- Separates planning and execution
- Well-supported, open-source
- Modular



HashiCorp
Terraform

Some Terraform basics

How it knows what to provision



Desired state looks like this

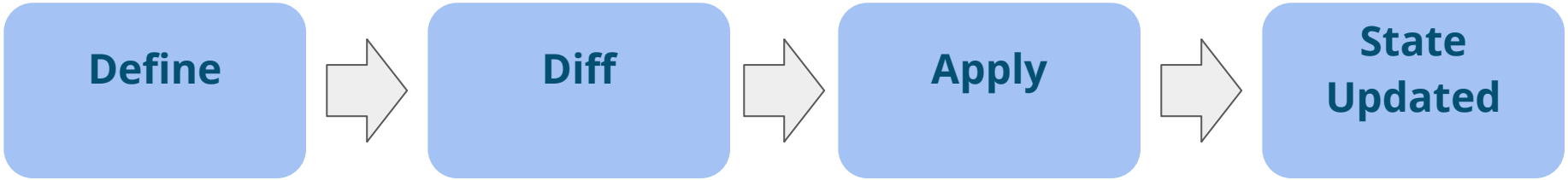
```
provider "aws" {
  access_key = "ACCESS_KEY_HERE"
  secret_key = "SECRET_KEY_HERE"
  region     = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-2757f631"
  instance_type = "t2.micro"
}
```

Actual state looks like this

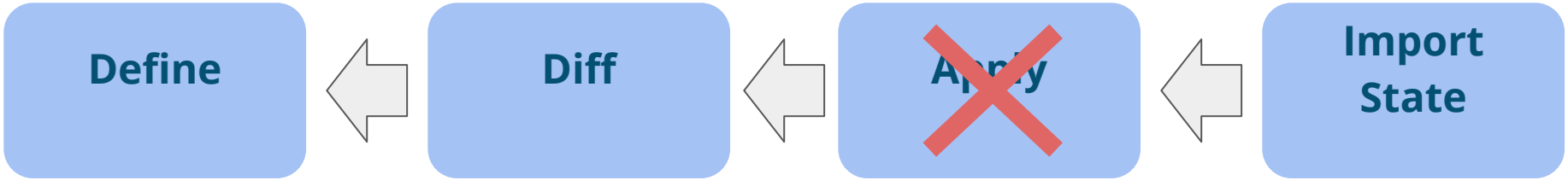
```
$ terraform show
aws_instance.example:
  id = i-32cf65a8
  ami = ami-2757f631
  availability_zone = us-east-1a
  instance_state = running
  instance_type = t2.micro
  private_ip = 172.31.30.244
  public_dns = ec2-52-90-212-55.compute-1.amazonaws.com
  public_ip = 52.90.212.55
  subnet_id = subnet-1497024d
  vpc_security_group_ids.# = 1
  vpc_security_group_ids.3348721628 = sg-67652003
```

Prototyping

Greenfield approach



Reverse engineering approach



Refactor to use variables

Hardcoded

```
resource "aws_vpc" "default" {
  cidr_block = "10.0.0.0/24"

  tags {
    "Name" = "hcgov-sls-prod"
  }
}
```

Variables

```
variable "vpc_name" {
  default = "hcgov-sls-prototype"
}

variable "vpc_cidr" {
  default = "10.0.1.0/24"
}

resource "aws_vpc" "default" {
  cidr_block = "${var.vpc_cidr}"

  tags {
    "Name" = "${var.vpc_name}"
  }
}
```

Testing

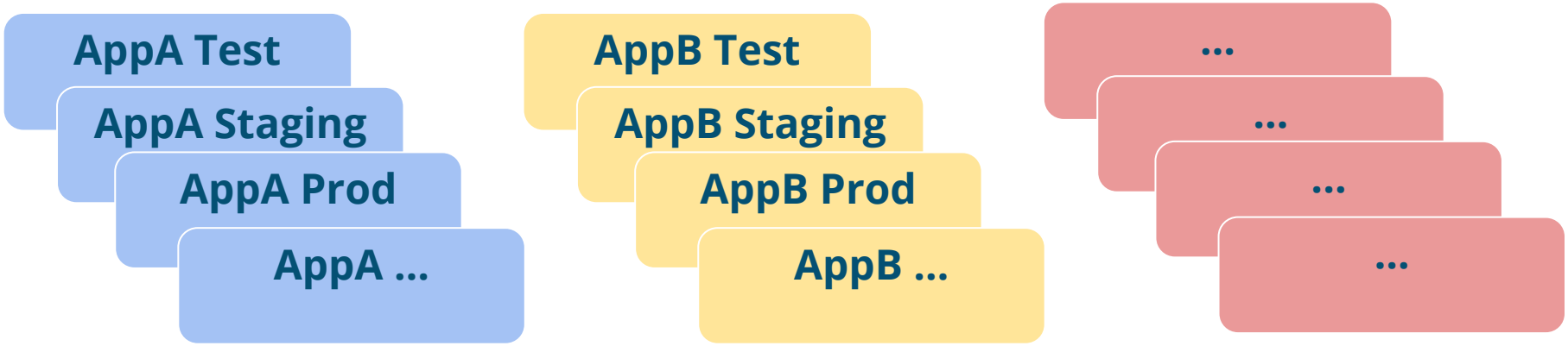
1. Successfully provision a new VPC
2. Application functional
    a. Passes health checks
    b. Passes smoke testing
3. Infrastructure security scan
    a. AWS Trusted Advisor

End result

- A configuration file (.tf) that represents one complete vpc configuration
- A state file (.tfstate) that represents one existing vpc

Design

How can we design this for reuse?

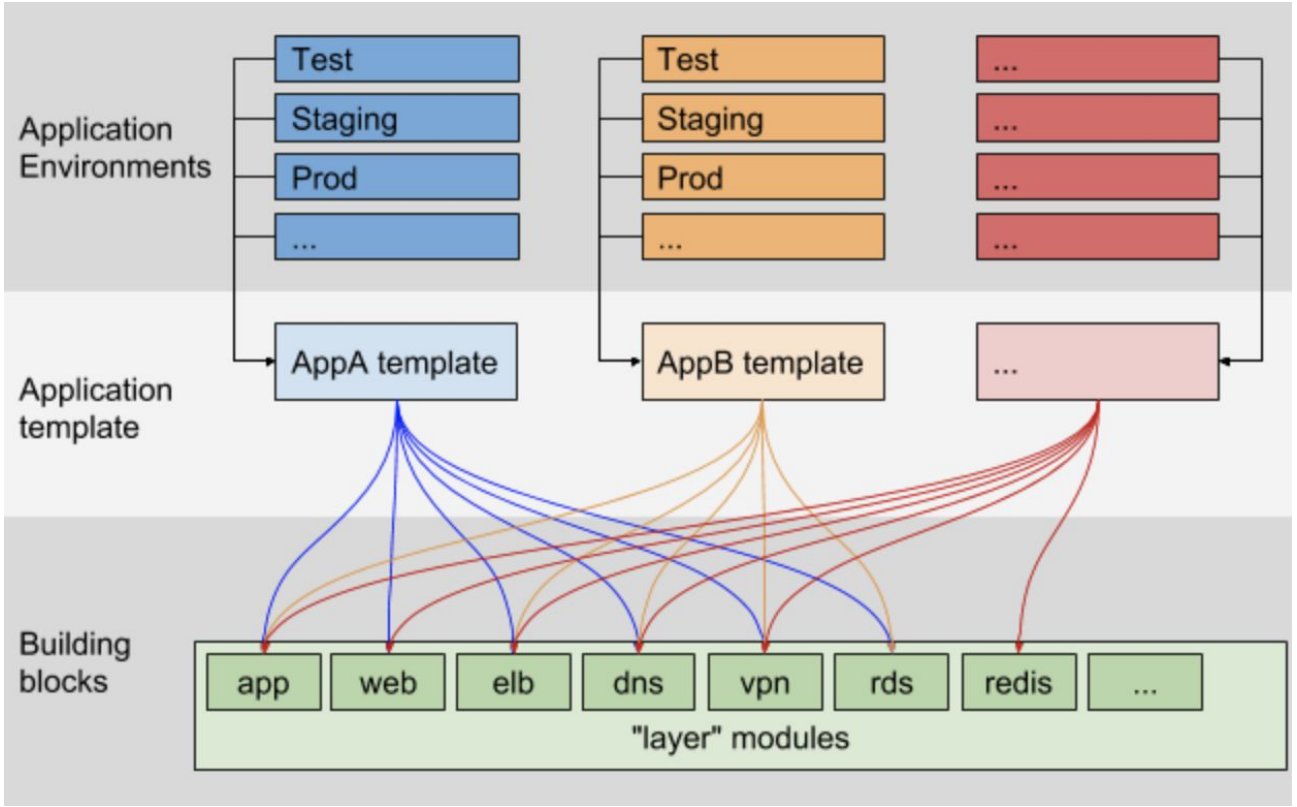


Existing design

Variable inputs

Assemble building blocks

Building blocks

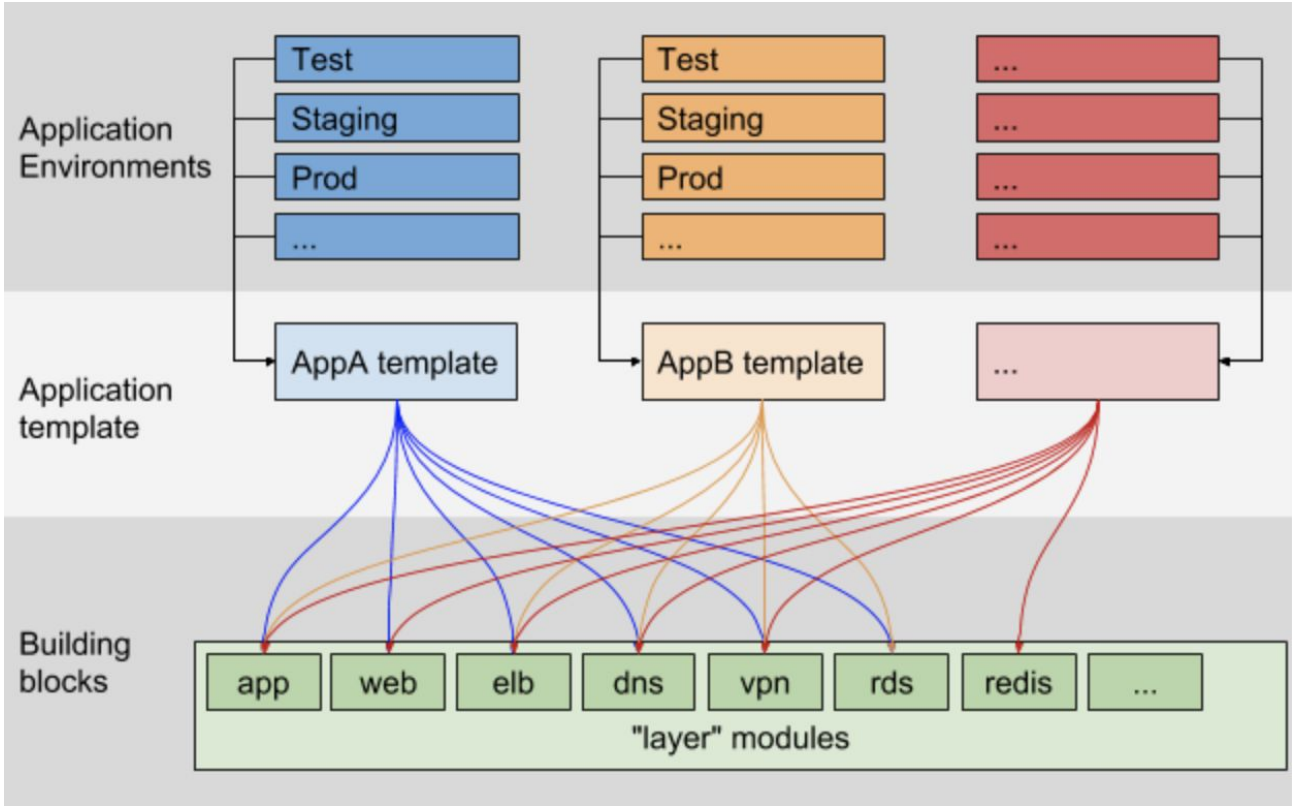


Implementation

Build new VPC's & cutover traffic

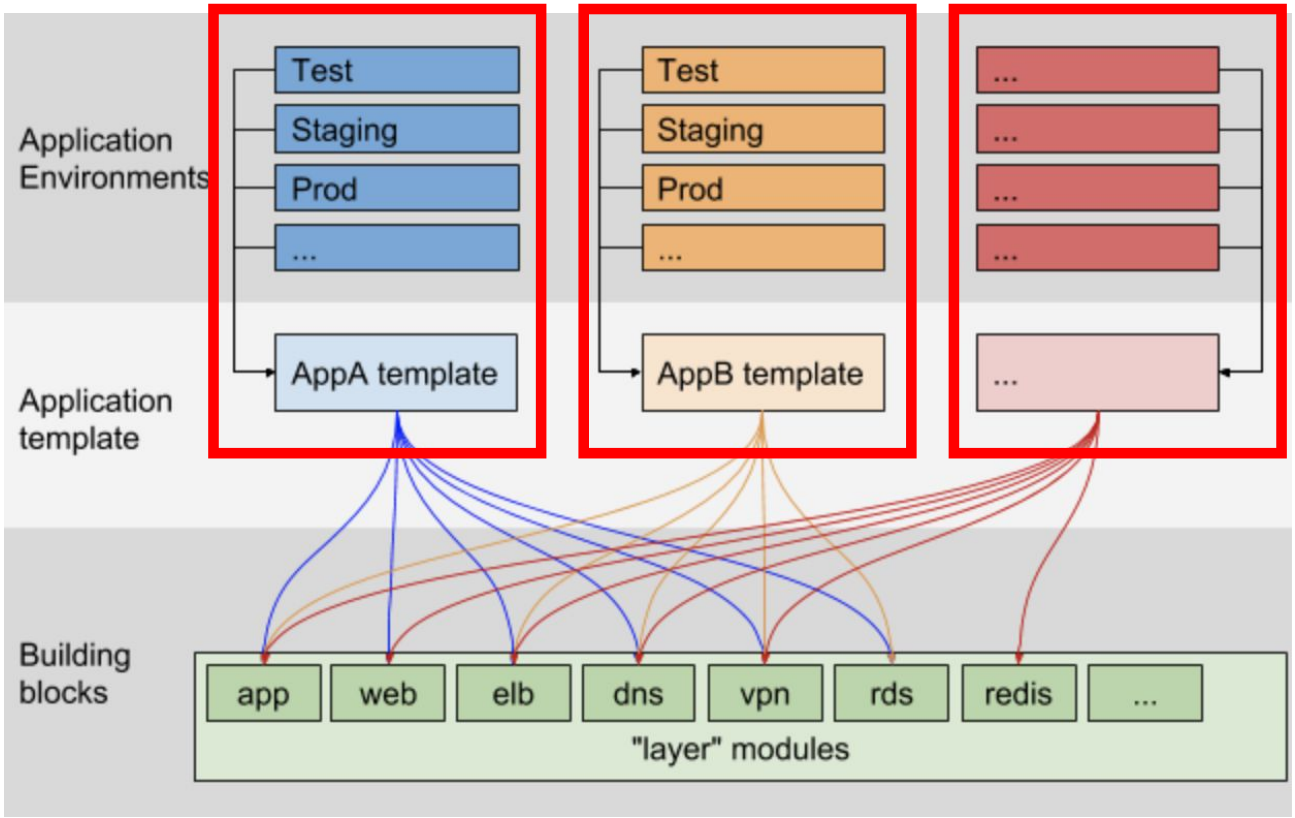
Learnings

Use shared modules sparingly



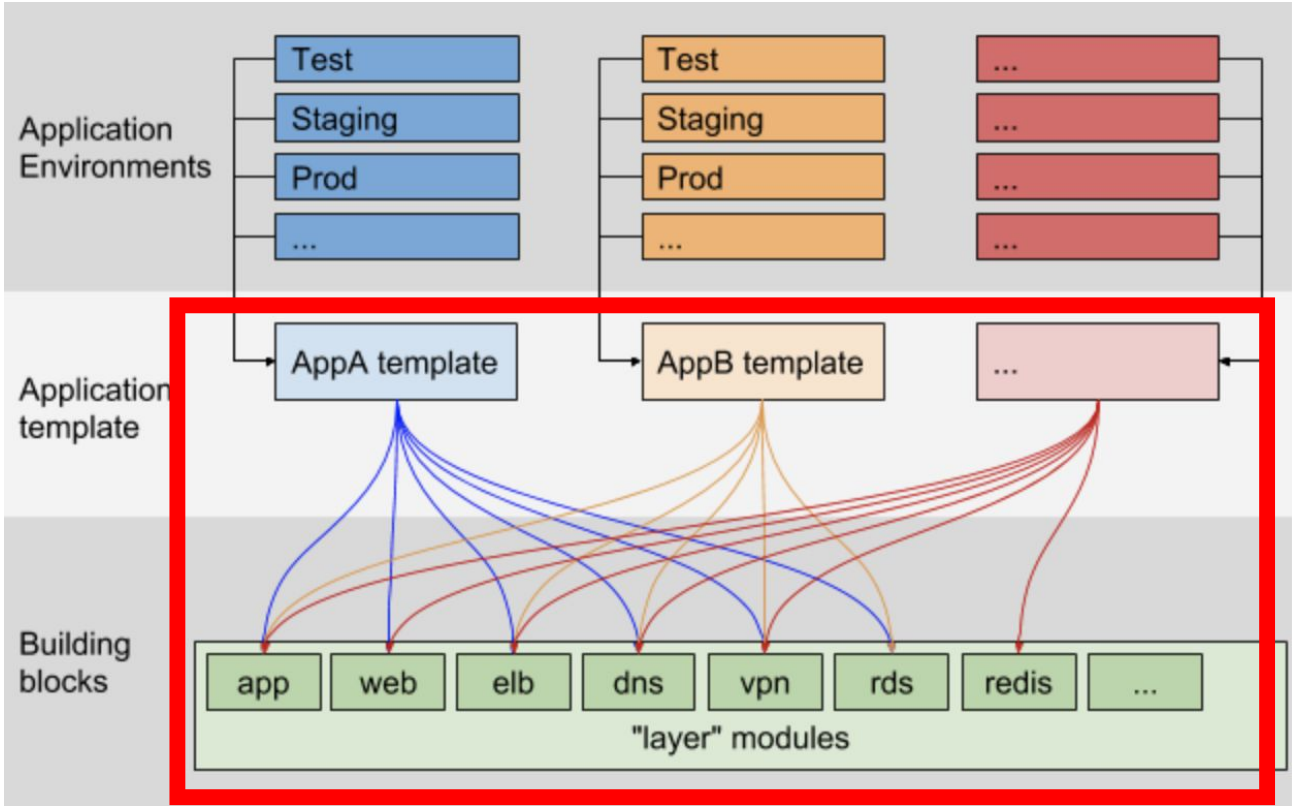
Use shared modules sparingly

Sharing modules ***within*** applications worked well



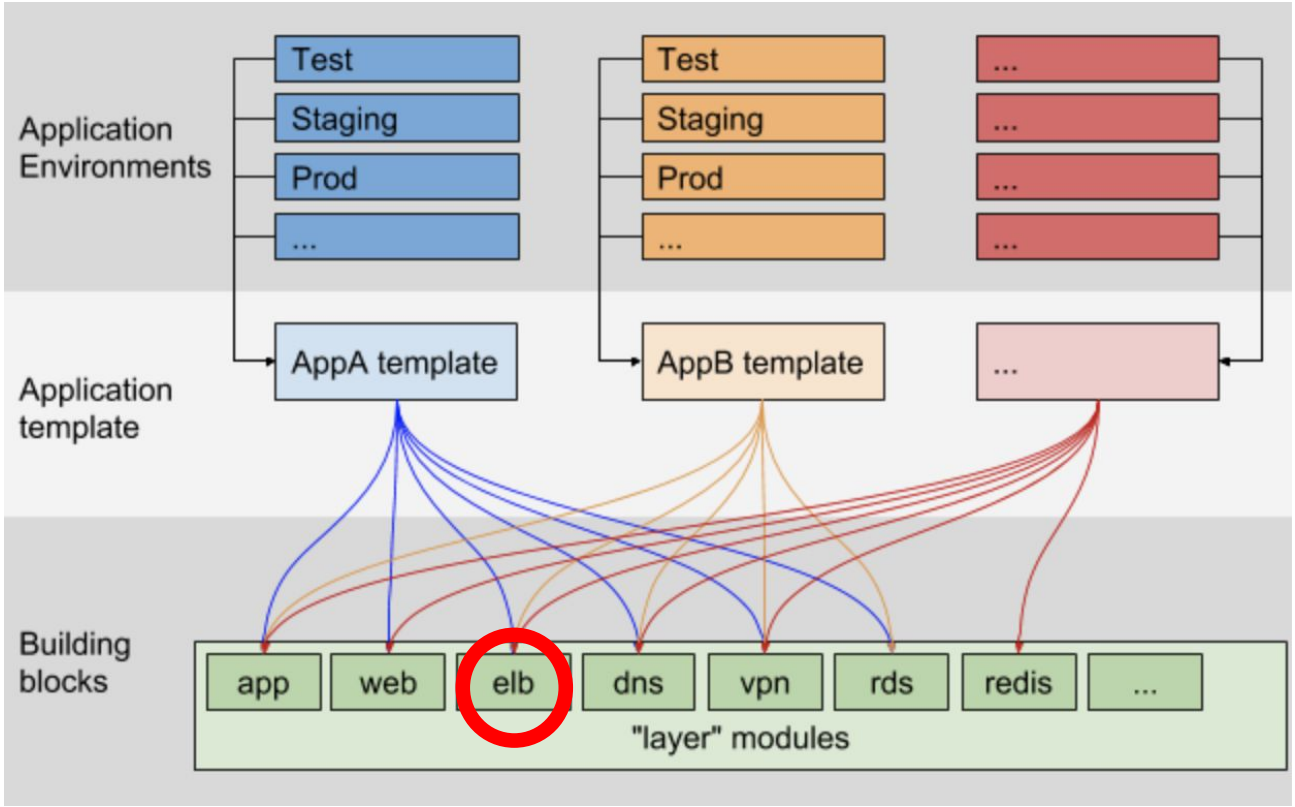
Use shared modules sparingly

Sharing modules ***across*** applications did not work well

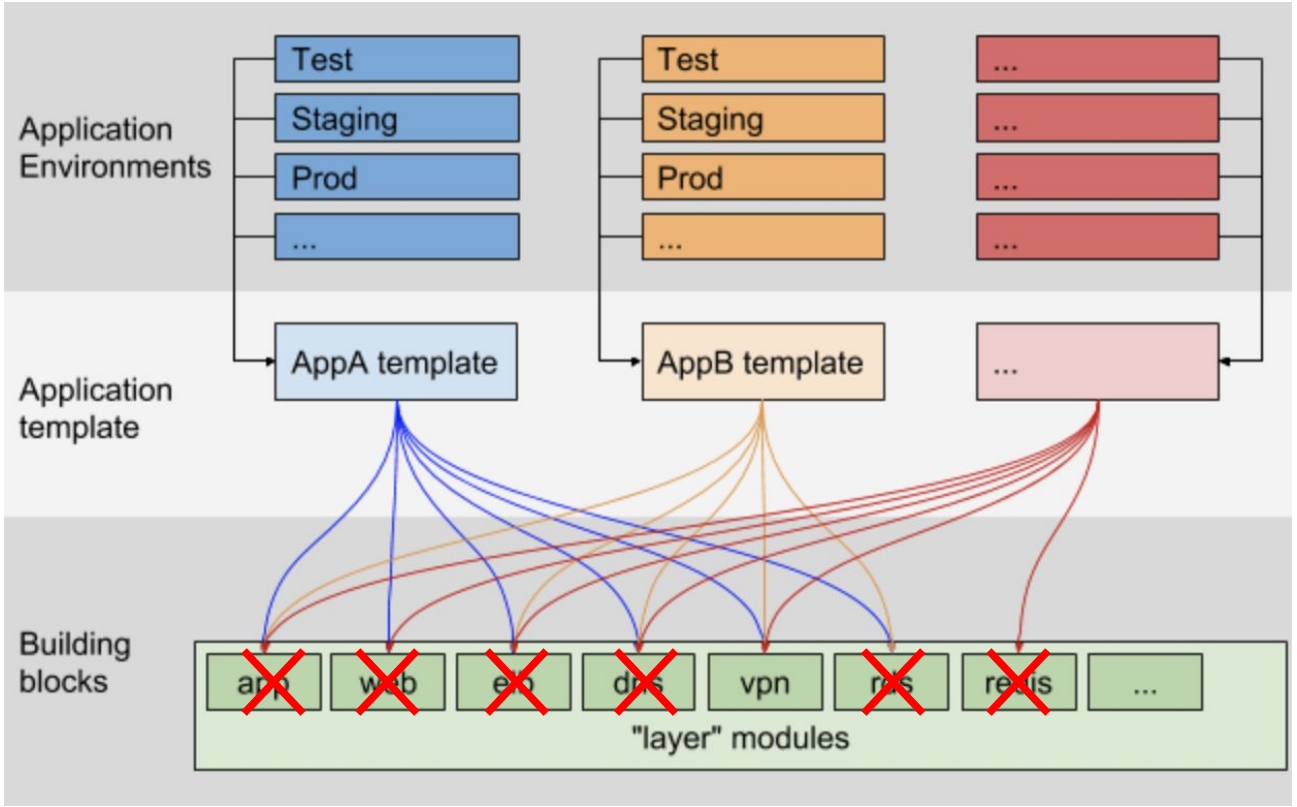


Use shared modules sparingly

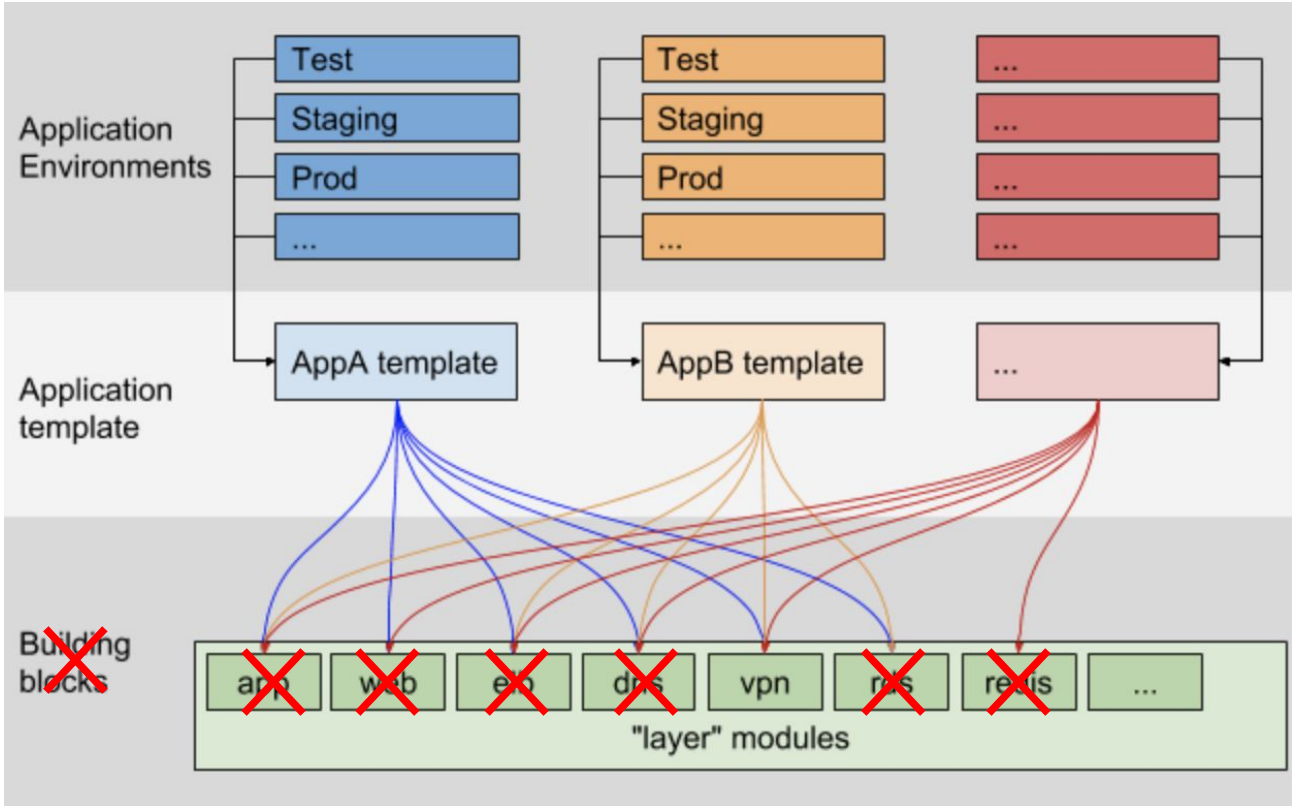
Change the **Elastic Load Balancer** module



Use shared modules sparingly



Use shared modules sparingly



constants

Migrating infrastructure in place

It's possible, but time consuming

Importing existing state

- Native **`terraform import`** CLI utility
  - Only imports one resource at a time
  - Requires manually finding each resource id relevant to a particular vpc
- Third party open source **`terraforming`** CLI
  - Imports all resources in a region
  - Cannot narrow scope to a specific vpc

Lock resources to a particular terraform version

```
terraform {
  backend "s3" {
    bucket         = "aws-xxx-xxx-xxx-xxx-us-east-1"
    key            = "hcgov-sls-prod/terraform/terraform.tfstate"
    region         = "us-east-1"
    dynamodb_table = "tf_lock"
  }

  required_version = "~> 0.11.7"
}
```

Terraform needs to be managed in CI/CD

Otherwise:

- Risk losing internet connection in mid-apply
- No record of who changed what when
- Developers bump versions unintentionally

Semantically version modules with git tags

Good

```
module "jump" {
  source = "git@github.com:CMSgov/terraform//jump?ref=1.0.5"
```

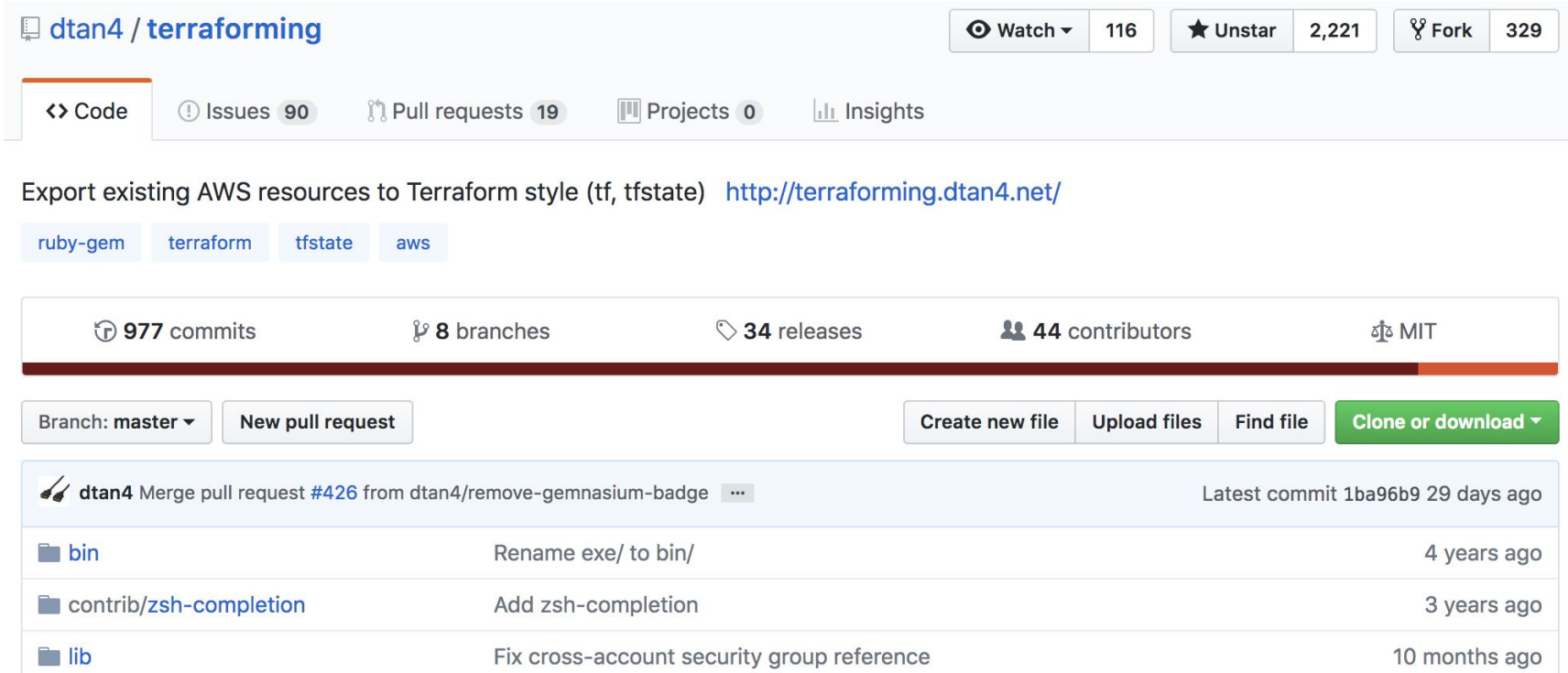
Bad

```
module "jump" {
  source = "git@github.com:CMSgov/terraform//jump?ref=598a8ebe3e428b37e806668995d7ff5ac20f1d7a"
```

Terraform utilities

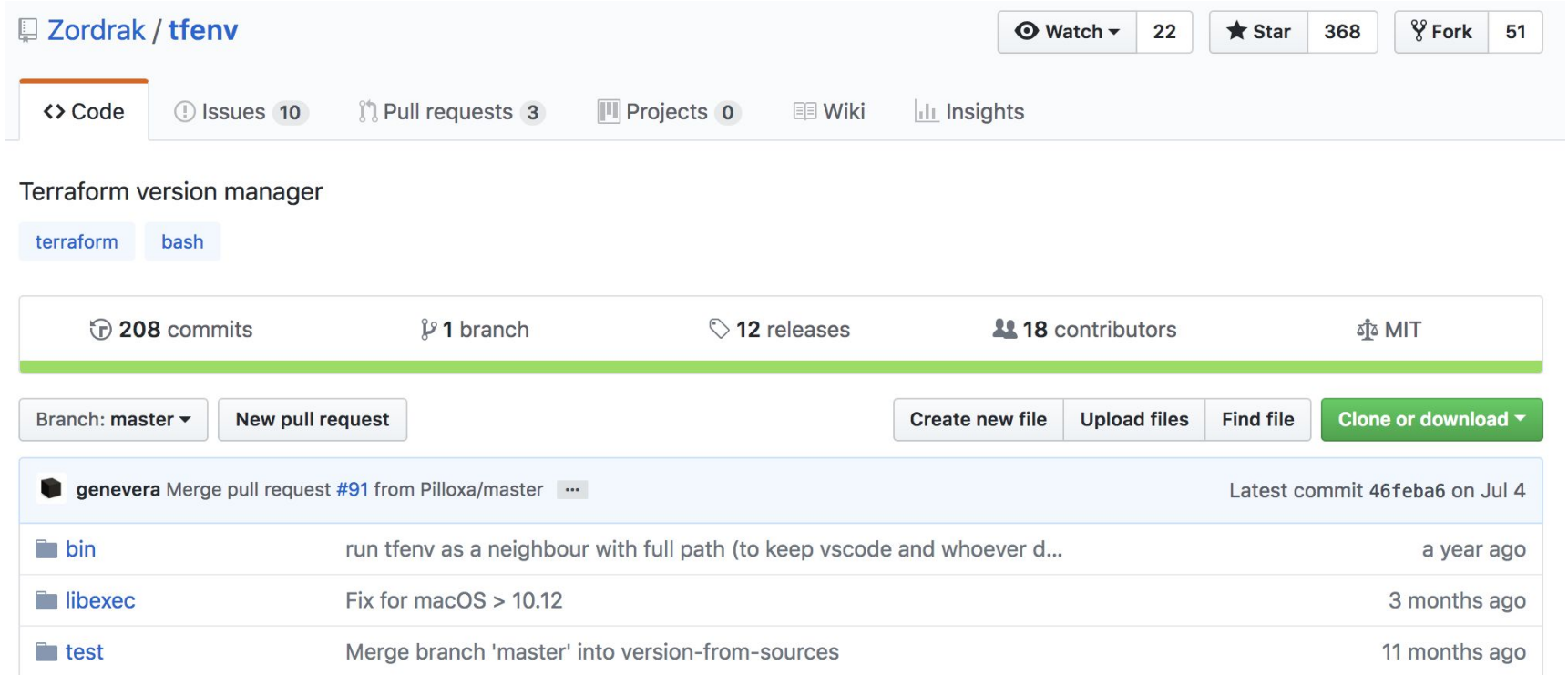
terraforming

Export existing AWS resources to Terraform



tfenv

Terraform version manager inspired by rbenv



terraform fmt

Before

```
resource "aws_cloudwatch_metric_alarm"
  alarm_name = "ec2-${var.vpc_name}-we
  count = "${var.monitor_web_cpu == "t
  comparison_operator = "GreaterThanOr
  namespace  = "AWS/EC2"
  metric_name = "CPUUtilization"
  statistic = "Average"
  unit = "Percent"
  # Trigger alarm on > 90% CPU for 60s
  threshold = "60"
  period = "60"
  evaluation_periods = "2"
  # Alarms are defined
  alarm_actions = ["${local.cloudwatch
  ok_actions    = ["${local.cloudwatch
  dimensions {
    AutoScalingGroupName = "${module.a
  }
}
```

After

```
resource "aws_cloudwatch_metric_alarm"
  alarm_name          = "ec2-${var.vpc_
  count               = "${var.monitor_
  comparison_operator = "GreaterThanOrE
  namespace           = "AWS/EC2"
  metric_name         = "CPUUtilization
  statistic           = "Average"
  unit                = "Percent"

  # Trigger alarm on > 90% CPU for 60s
  threshold           = "60"
  period              = "60"
  evaluation_periods = "2"

  # Alarms are defiend
  alarm_actions = ["${local.cloudwatch_
  ok_actions    = ["${local.cloudwatch_

  dimensions {
    AutoScalingGroupName = "${module.ap
```

terraform-docs

Generate docs from terraform modules

Inputs

Name	Description	Default	Required
subnet_ids	a comma-separated list of subnet IDs	-	yes

Outputs

Name	Description
vpc_id	The VPC ID.

Thank you

@monaghan_a_gram