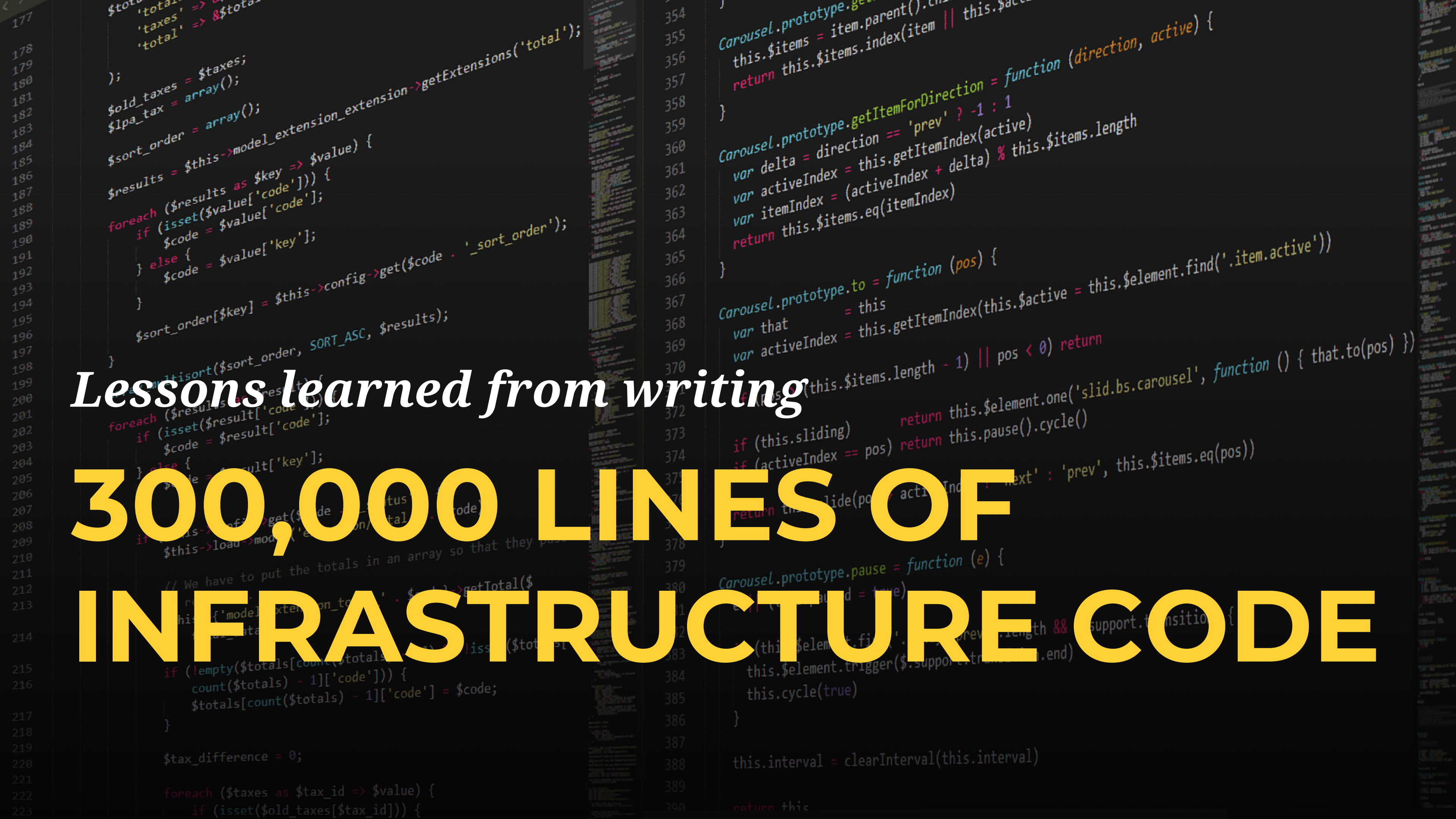


Lessons learned from writing

300,000 LINES OF INFRASTRUCTURE CODE

It’s time for a confession:

DevOps is still in the stone ages

We are trying to build this…

Using this.

If you just read the headlines, it all *sounds* so cutting edge…

Kubernetes, Docker, serverless, microservices, infrastructure as code, distributed tracing, big data systems, data warehouses, data lakes, chaos engineering, zero-trust architecture, streaming architecture, immutable infrastructure, service discovery, service meshes, NoSQL, NewSQL, ChatOps, HugOps, NoOps, DevSecOpsLeanSREAgileWTFBBQ, ...

But to me, it doesn’t *feel* cutting edge. It feels more like…

#thisisdevops

#thisisdevops

#thisisdevops

#thisisdevops

Here’s something we don’t admit often enough:

Building production-grade infrastructure is hard.

And stressful.

And time consuming.

Some rough numbers:

Production-grade infrastructure

Project	Examples	Time estimate
Managed service	ECS, ELB, RDS, ElastiCache	1 – 2 weeks
Distributed system (stateless)	nginx, Node.js app, Rails app	2 – 4 weeks
Distributed system (stateful)	Elasticsearch, Kafka, MongoDB	2 – 4 months
Entire cloud architecture	Apps, DBs, CI/CD, monitoring, etc.	6 – 24 months

Fortunately, it’s getting a little bit better

```
$tot
	'tota
	'taxes' => &$tota
	'total' => &$tota
);

$old_taxes = $taxes;
$lpa_tax = array();

$sort_order = array();

$results = $this->model_extension_extension->getExtensions('total');

foreach ($results as $key => $value) {
	if (isset($value['code'])) {
		$code = $value['code'];
	} else {
		$code = $value['key'];
	}

	$sort_order[$key] = $this->config->get($code . '_sort_order');
}

array_multisort($sort_order, SORT_ASC, $results);

foreach ($results as $result) {
	if (isset($result['code'])) {
		$code = $result['code'];
	} else {
		$code = $result['key'];
	}

	if ($this->config->get($code . '_status')) {
		$this->load->model('extension/total/' . $code);

		// We have to put the totals in an array so that they pass
		by reference.
		$this->{'model_extension_total_' . $code}->getTotal($
		total_data);

		if (!empty($totals[count($totals) - 1]) && !isset($totals[
			count($totals) - 1]['code'])) {
			$totals[count($totals) - 1]['code'] = $code;

		$tax_difference = 0;

		foreach ($taxes as $tax_id => $value) {
```

```
  }

  Carousel.prototype.ge
    this.$items = item.parent().ch
    return this.$items.index(item || this.$act
  }

  Carousel.prototype.getItemForDirection = function (direction, active) {
    var delta = direction == 'prev' ? -1 : 1
    var activeIndex = this.getItemIndex(active)
    var itemIndex = (activeIndex + delta) % this.$items.length
    return this.$items.eq(itemIndex)
  }

  Carousel.prototype.to = function (pos) {
    var that        = this
    var activeIndex = this.getItemIndex(this.$active = this.$element.find('.item.active'))

    if (pos > (this.$items.length - 1) || pos < 0) return

    if (this.sliding)       return this.$element.one('slid.bs.carousel', function () { that.to(pos) })
    if (activeIndex == pos) return this.pause().cycle()

    return this.slide(pos > activeIndex ? 'next' : 'prev', this.$items.eq(pos))
  }

  Carousel.prototype.pause = function (e) {
    e || (this.paused = true)

    if (this.$element.find('.next, .prev').length && $.support.transition) {
      this.$element.trigger($.support.transition.end)
      this.cycle(true)
    }

    this.interval = clearInterval(this.interval)
```

One trend I love: manage (almost) everything as code

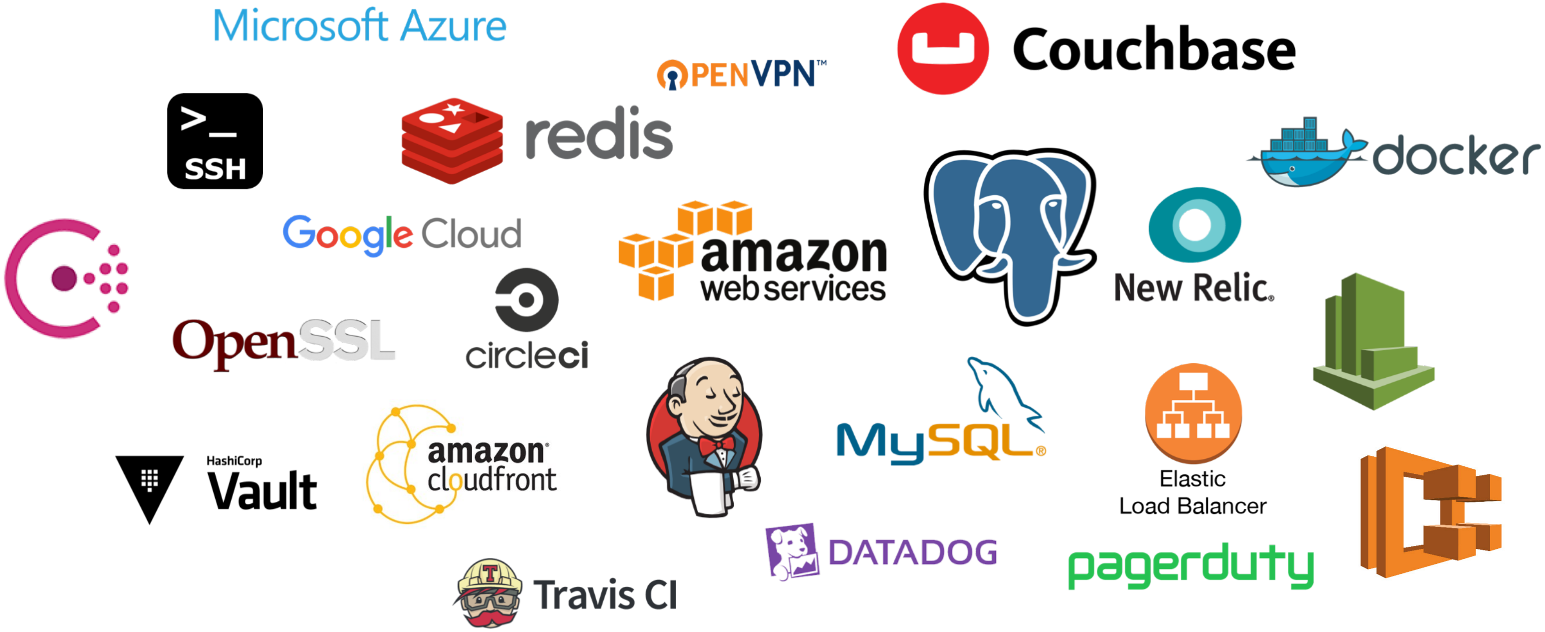
Manual provisioning	→	Infrastructure as code
Manual server config	→	Configuration management
Manual app config	→	Configuration files
Manual builds	→	Continuous integration
Manual deployment	→	Continuous delivery
Manual testing	→	Automated testing
Manual DBA work	→	Schema migrations
Manual specs	→	Automated specs (BDD)

The benefits of code:

1. Automation
2. Version control
3. Code review
4. Testing
5. Documentation
6. Reuse

At Gruntwork, we’ve created a reusable library of infrastructure code

Primarily written in Terraform, Go, Python, and Bash



Off-the-shelf, battle-tested solutions for AWS, Docker, VPCs, VPN, MySQL, Postgres, Couchbase, ElasticSearch, Kafka, ZooKeeper, Monitoring, Alerting, secrets management, CI, CD, DNS, …

3+ years of development.
300,000+ lines of code.

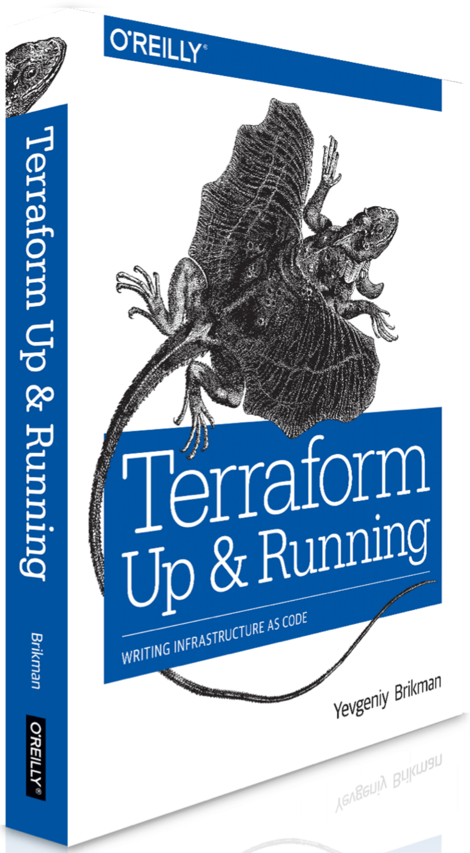
In this talk, I’ll share what we learned along the way!

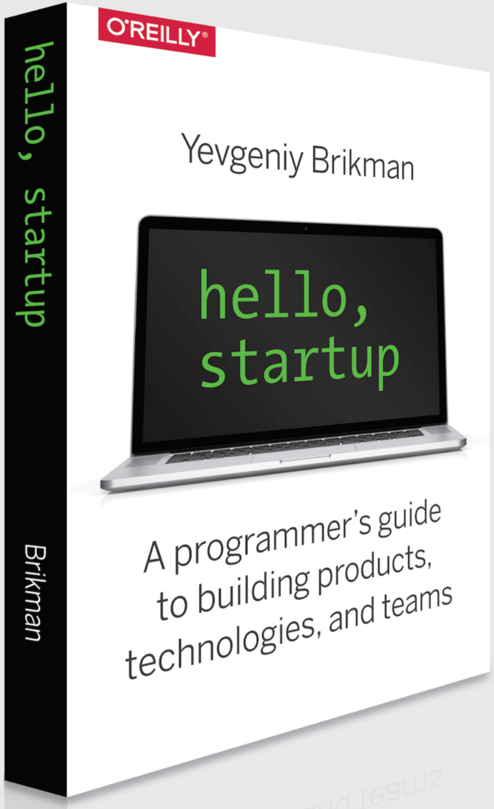
I’m Yevgeniy Brikman

ybrikman.com

gruntwork.io

Co-founder of Gruntwork


O'REILLY
Terraform Up & Running
Terraform
Up & Running
WRITING INFRASTRUCTURE AS CODE
Yevgeniy Brikman
Brikman
O'REILLY


O'REILLY
hello, startup
Yevgeniy Brikman
hello,
startup
A programmer's guide
to building products,
technologies, and teams
Brikman

Author

Outline

1. Checklist
2. Tools
3. Modules
4. Tests
5. Releases

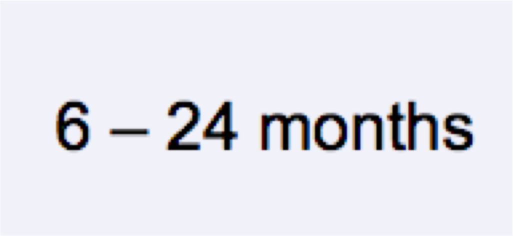
Outline

1. Checklist
2. Tools
3. Modules
4. Tests
5. Releases

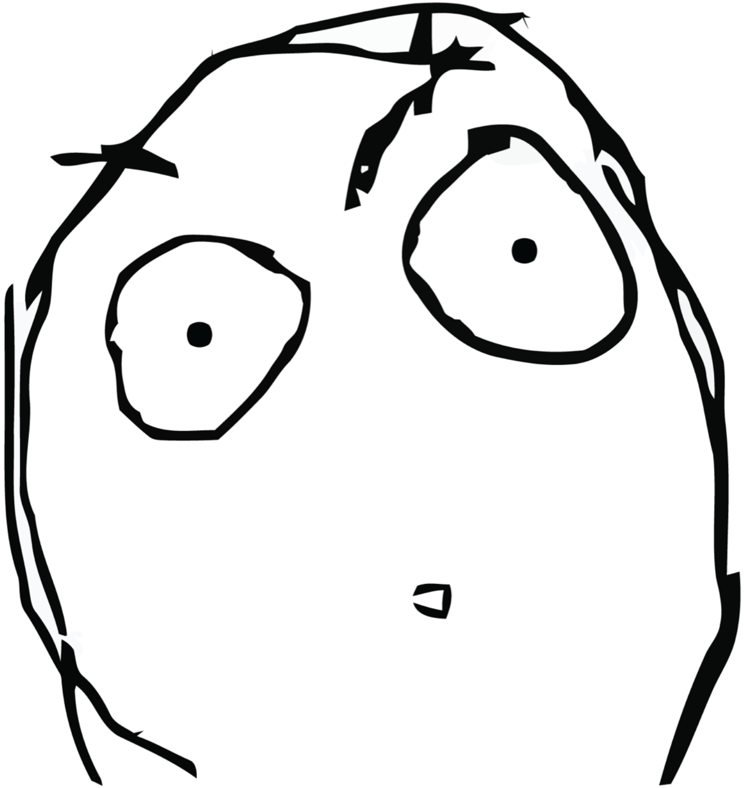
DevOps newbies are always shocked by these numbers:

Production-grade infrastructure

Project	Examples	Time estimate
Managed service	ECS, ELB, RDS, ElastiCache	1 – 2 weeks
Distributed system (stateless)	nginx, Node.js app, Rails app	2 – 4 weeks
Distributed system (stateful)	Elasticsearch, Kafka, MongoDB	2 – 4 months
Entire cloud architecture	Apps, DBs, CI/CD, monitoring, etc.	6 – 24 months



6 – 24 months

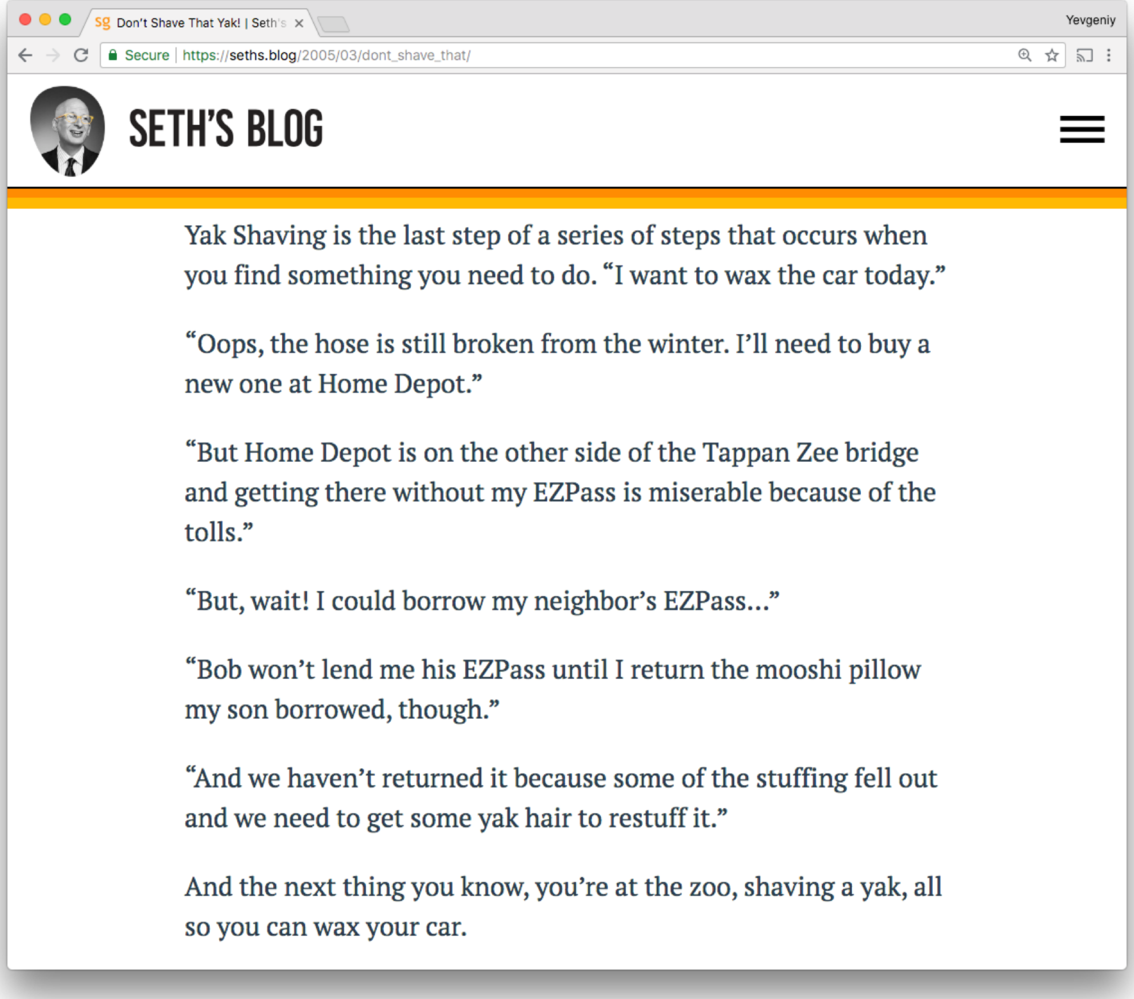


How can it possibly take that long??

Two main reasons:

Reason it takes so long #1:
Yak shaving

Yak shaving: a seemingly endless series of small tasks you have to do before you can do what you actually want.



Don't Shave That Yak! | Seth's
Yevgeniy
Secure
https://seths.blog/2005/03/dont_shave_that/
SETH'S BLOG

Yak Shaving is the last step of a series of steps that occurs when you find something you need to do. “I want to wax the car today.”

“Oops, the hose is still broken from the winter. I’ll need to buy a new one at Home Depot.”

“But Home Depot is on the other side of the Tappan Zee bridge and getting there without my EZPass is miserable because of the tolls.”

“But, wait! I could borrow my neighbor’s EZPass…”

“Bob won’t lend me his EZPass until I return the mooshi pillow my son borrowed, though.”

“And we haven’t returned it because some of the stuffing fell out and we need to get some yak hair to restuff it.”

And the next thing you know, you’re at the zoo, shaving a yak, all so you can wax your car.

Reason it takes so long #2:
It’s a long checklist!

Introducing:

The production-grade infrastructure checklist

Production-grade infrastructure checklist, part 1/4

Task	Description	Example tools
Install	Install the software binaries and all dependencies.	Bash, Chef, Ansible, Puppet
Configure	Configure the software at runtime: e.g., configure port settings, file paths, users, leaders, followers, replication, etc.	Bash, Chef, Ansible, Puppet
Provision	Provision the infrastructure: e.g., EC2 Instances, load balancers, network topology, security groups, IAM permissions, etc.	Terraform, CloudFormation
Deploy	Deploy the service on top of the infrastructure. Roll out updates with no downtime: e.g., blue-green, rolling, canary deployments.	Scripts, Orchestration tools (ECS, K8S, Nomad)

Production-grade infrastructure checklist, part 2/4

Task	Description	Example tools
Security	Encryption in transit (TLS) and on disk, authentication, authorization, secrets management, server hardening.	ACM, EBS Volumes, Cognito, Vault, CiS
Monitoring	Availability metrics, business metrics, app metrics, server, metrics, events, observability, tracing, alerting.	CloudWatch, DataDog, New Relic, Honeycomb
Logs	Rotate logs on disk. Aggregate log data to a central location.	CloudWatch Logs, ELK, Sumo Logic, Papertrail
Backup and restore	Make backups of DBs, caches, and other data on a scheduled basis. Replicate to separate region/account.	RDS, ElastiCache, ec2-snapper, Lambda

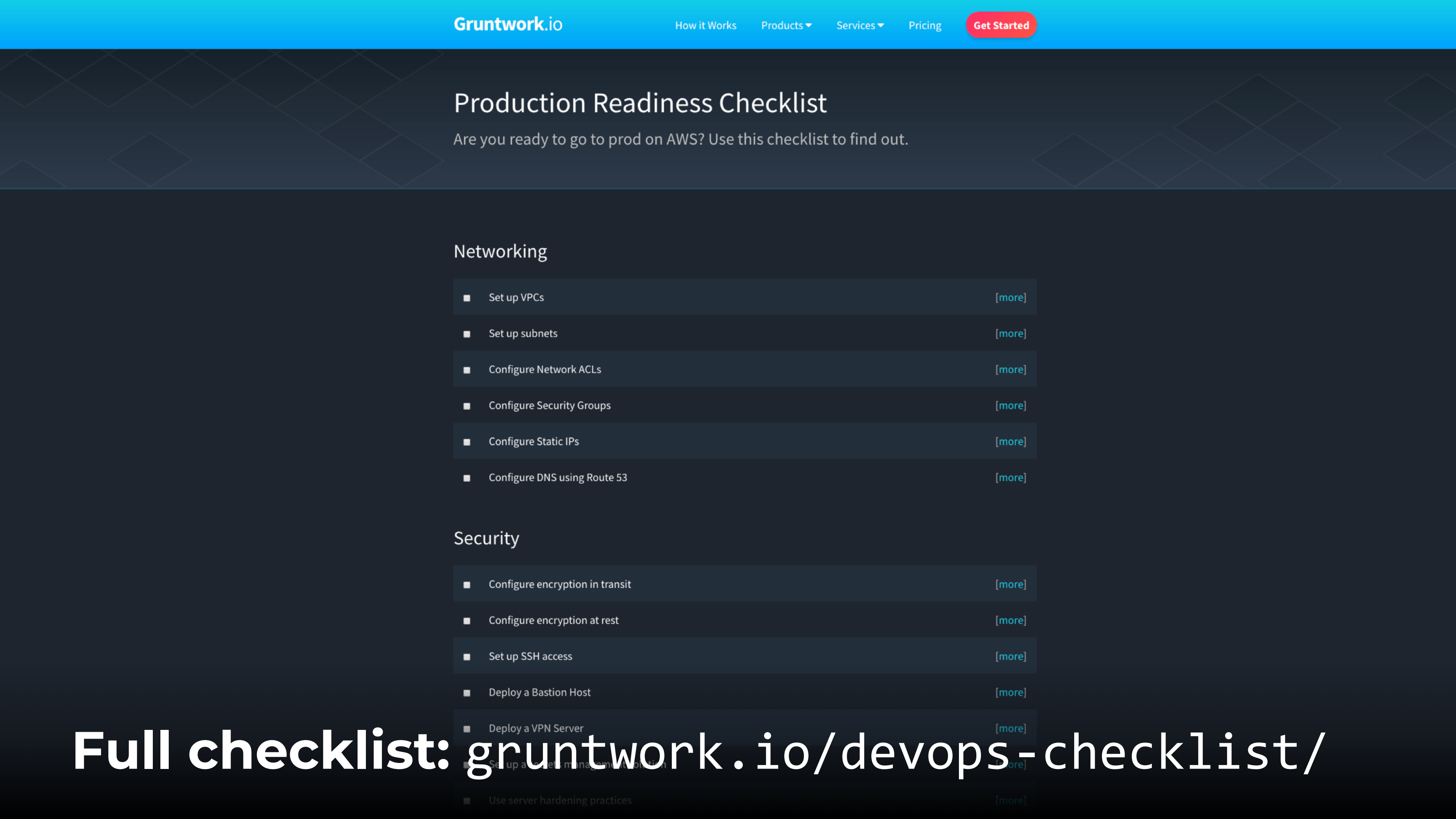
Production-grade infrastructure checklist, part 3/4

Task	Description	Example tools
Networking	VPCs, subnets, static and dynamic IPs, service discovery, service mesh, firewalls, DNS, SSH access, VPN access.	EIPs, ENIs, VPCs, NACLs, SGs, Route 53, OpenVPN
High availability	Withstand outages of individual processes, EC2 Instances, services, Availability Zones, and regions.	Multi AZ, multi-region, replication, ASGs, ELBs
Scalability	Scale up and down in response to load. Scale horizontally (more servers) and/or vertically (bigger servers).	ASGs, replication, sharding, caching, divide and conquer
Performance	Optimize CPU, memory, disk, network, GPU and usage. Query tuning. Benchmarking, load testing, profiling.	Dynatrace, valgrind, VisualVM, ab, Jmeter

Production-grade infrastructure checklist, part 4/4

Task	Description	Example tools
Cost optimization	Pick proper instance types, use spot and reserved instances, use auto scaling, nuke unused resources	ASGs, spot instances, reserved instances
Documentation	Document your code, architecture, and practices. Create playbooks to respond to incidents.	READMEs, wikis, Slack
Tests	Write automated tests for your infrastructure code. Run tests after every commit and nightly.	Terratest

Key takeaway: use a checklist to build production-grade infrastructure.



Production Readiness Checklist

Are you ready to go to prod on AWS? Use this checklist to find out.

Networking

- ■ Set up VPCs [more]
- ■ Set up subnets [more]
- ■ Configure Network ACLs [more]
- ■ Configure Security Groups [more]
- ■ Configure Static IPs [more]
- ■ Configure DNS using Route 53 [more]

Security

- ■ Configure encryption in transit [more]
- ■ Configure encryption at rest [more]
- ■ Set up SSH access [more]
- ■ Deploy a Bastion Host [more]
- ■ Deploy a VPN Server [more]
- ■ Set up a secrets management solution [more]
- ■ Use server hardening practices [more]

Full checklist: `gruntwork.io/devops-checklist/`

Outline

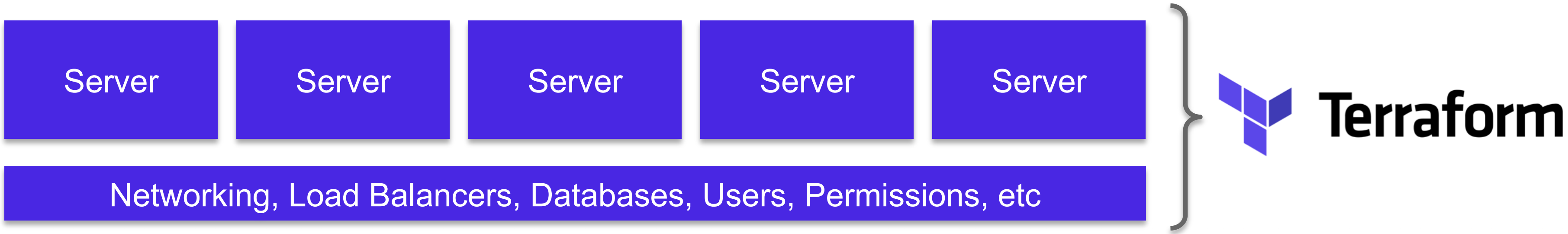
1. Checklist
2. Tools
3. Modules
4. Tests
5. Releases

What tools do you use to implement that checklist?

We prefer tools that:

1. Define infrastructure as code
2. Are open source & popular
3. Support multiple providers
4. Support reuse & composition
5. Require no extra infrastructure
6. Support immutable infrastructure

Here’s the toolset we’ve found most effective as of 2019:



Server

Server

Server

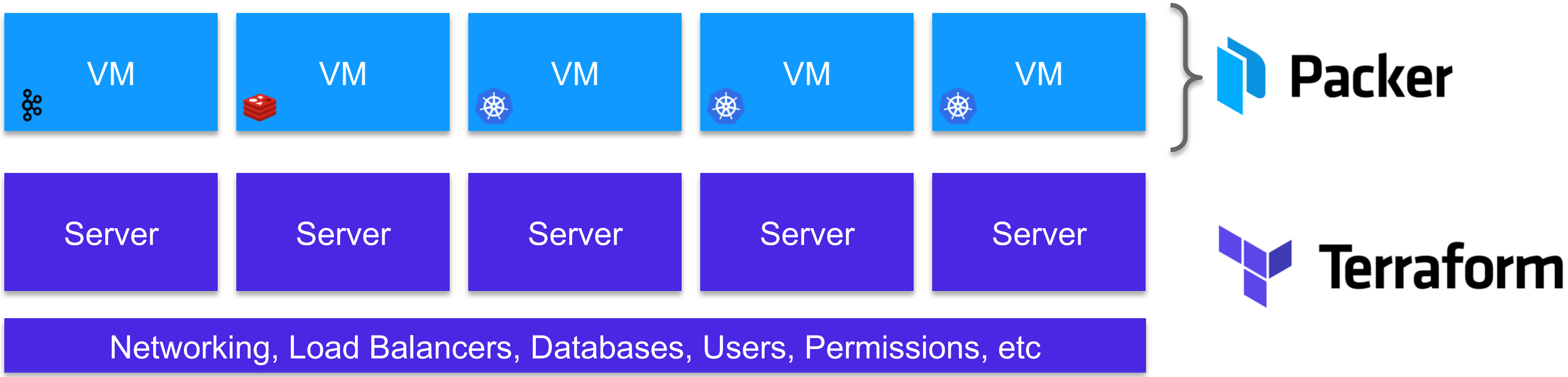
Server

Server

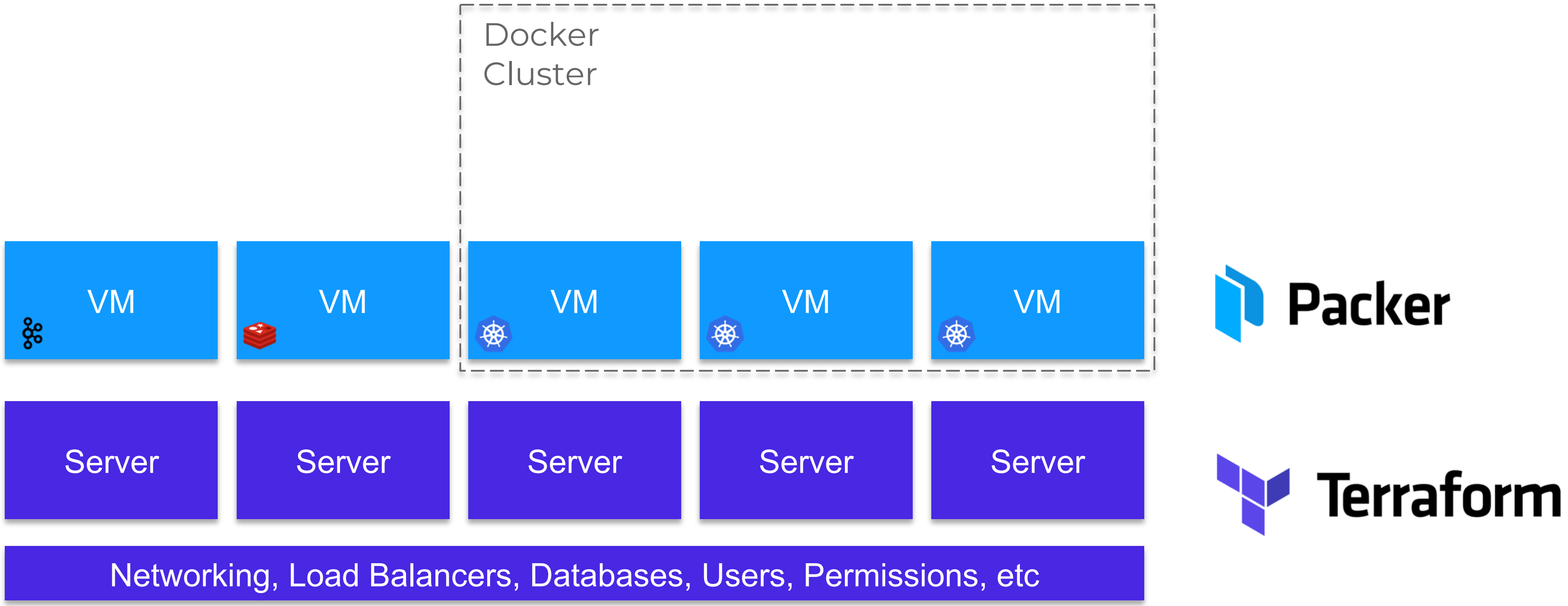
Networking, Load Balancers, Databases, Users, Permissions, etc

Terraform

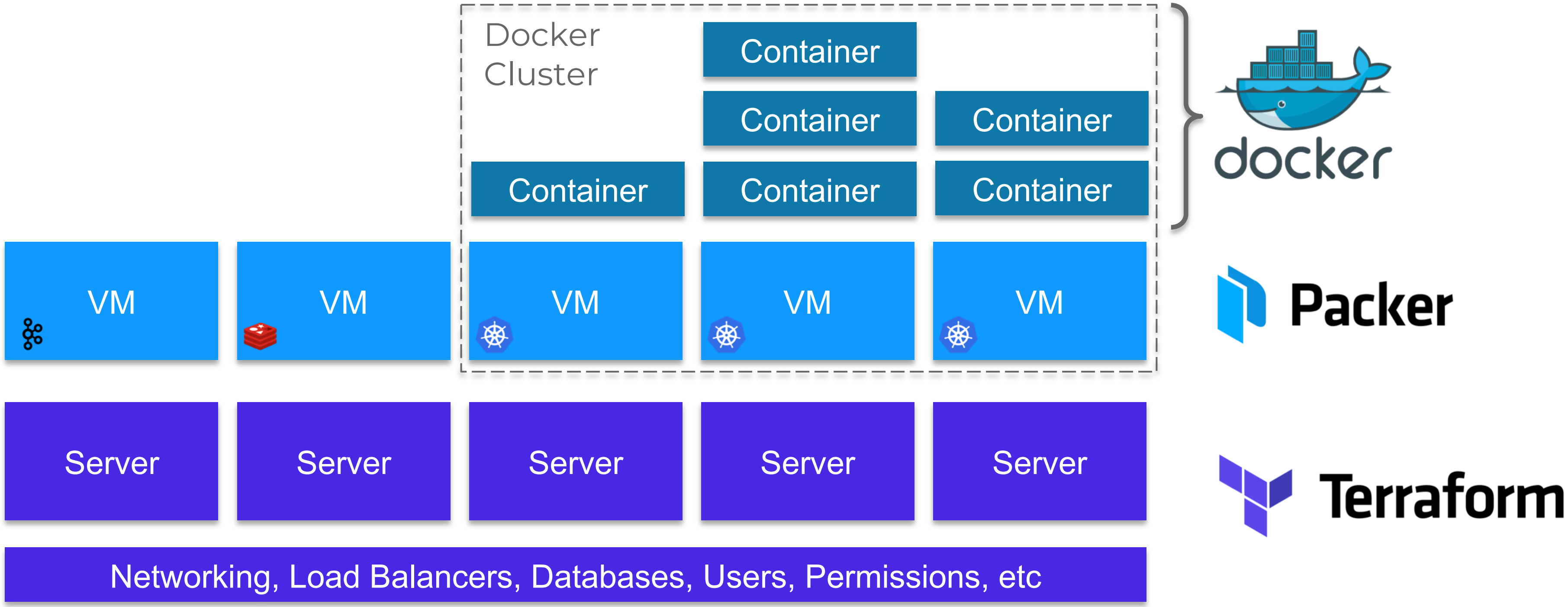
1. Deploy all the basic infrastructure using Terraform


VM
VM
VM
VM
VM
Packer
Server
Server
Server
Server
Server
Terraform
Networking, Load Balancers, Databases, Users, Permissions, etc

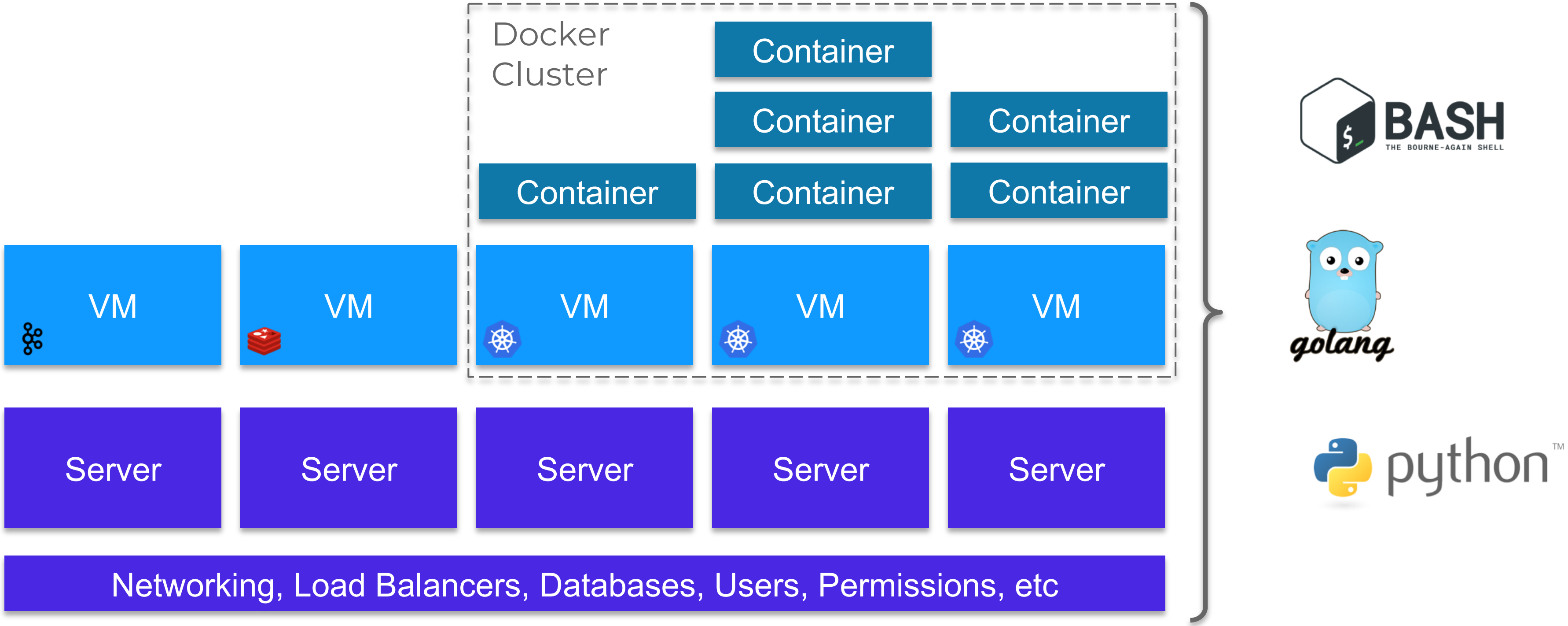
2. Configure the VMs using Packer



3. Some of the VMs form a cluster (e.g., ECS or Kubernetes cluster)

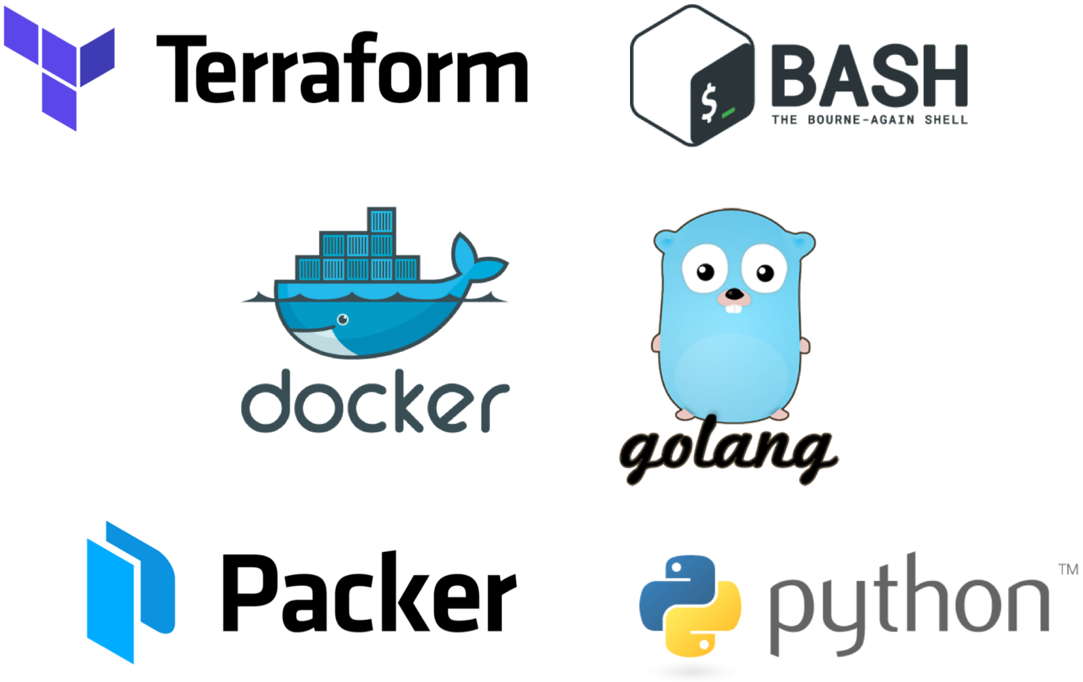


4. We use that Docker cluster to run Docker containers



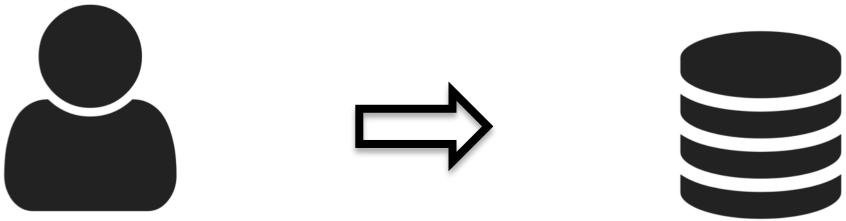
5. Under the hood: Bash, Go, and Python hold everything together

Important note:

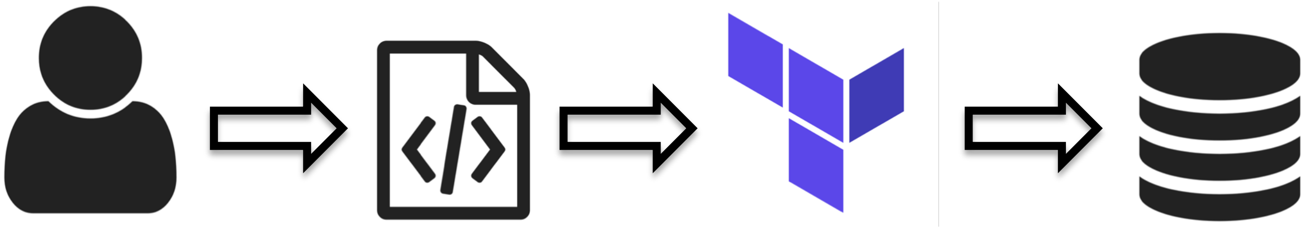


We find these tools useful...

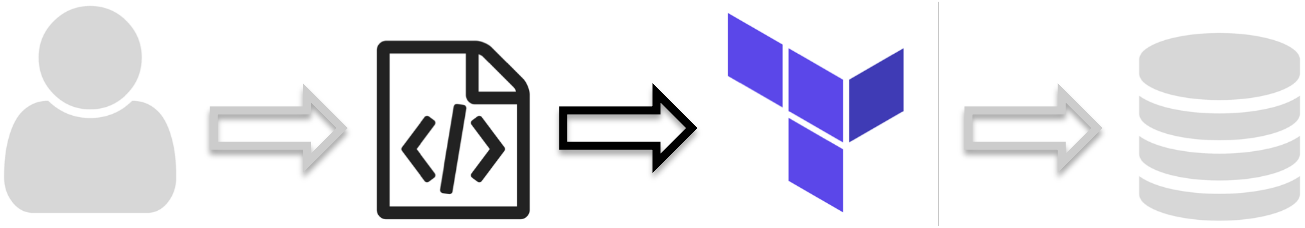
But tools are not enough.
You must change behavior too.



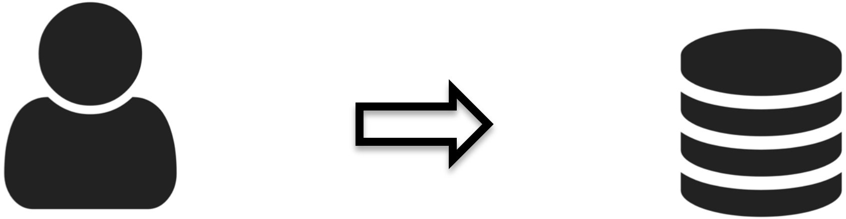
Old way: make changes directly and manually



New way: make changes indirectly and automatically

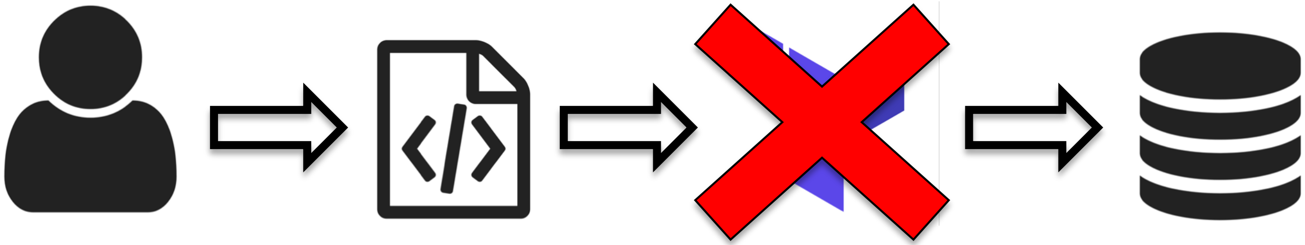


Learning these takes time

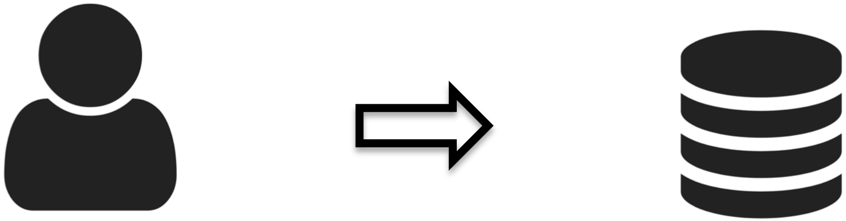


More time than making a change directly…

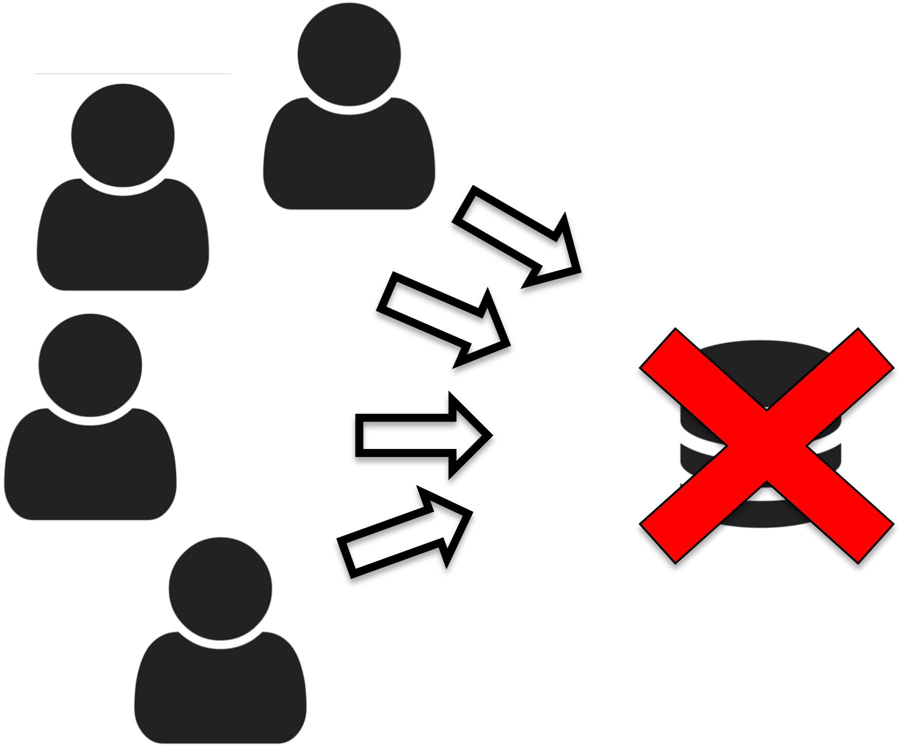
If you make changes manually,
the code will not reflect reality.



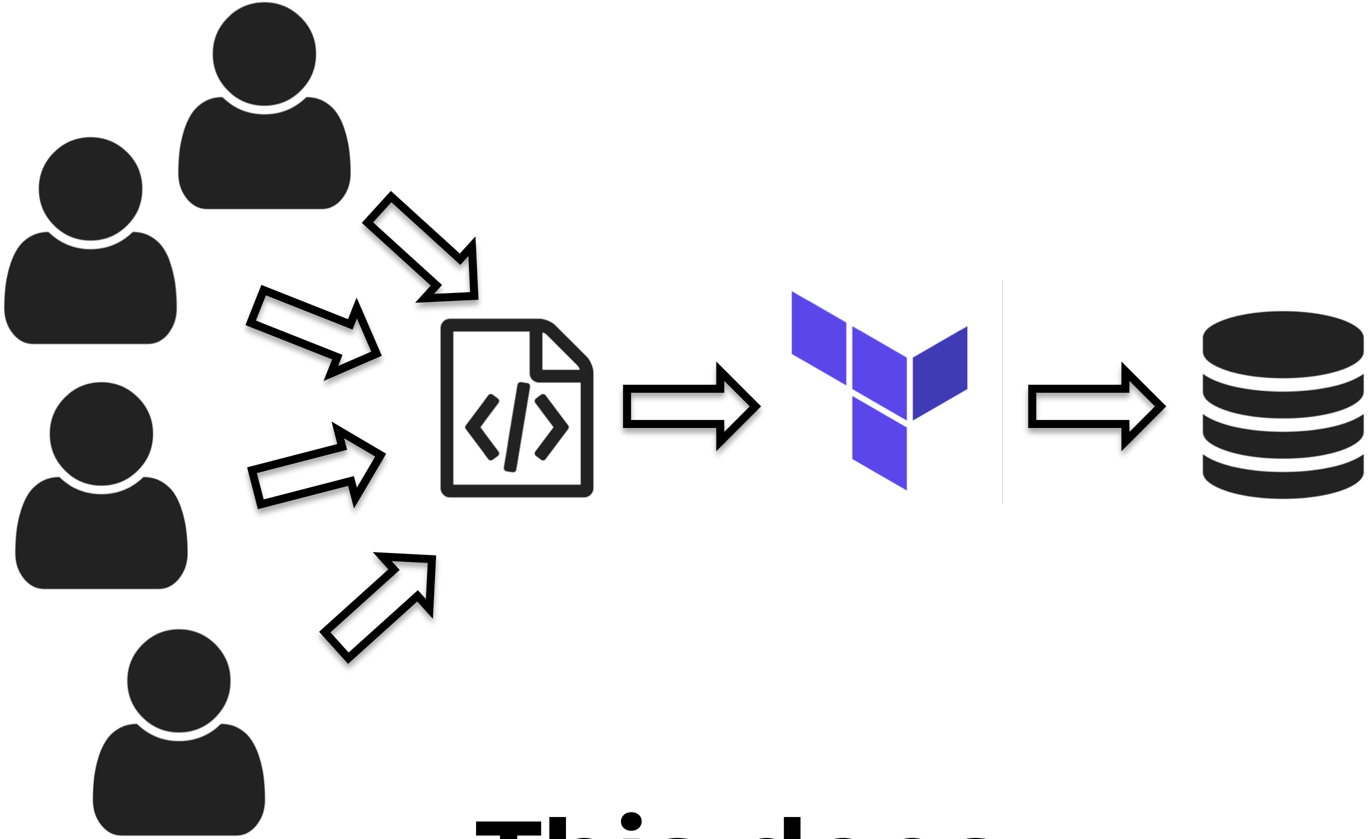
And the next person to try to use it will get errors



So then they’ll fall back and make manual changes



But making manual changes does not scale



This does

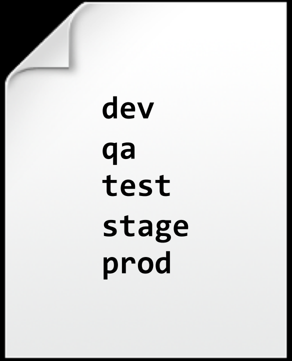
Key takeaway: tools are not enough. You also need to change behavior.

Outline

1. Checklist
2. Tools
3. **Modules**
4. Tests
5. Releases

```
dev
qa
test
stage
prod
```

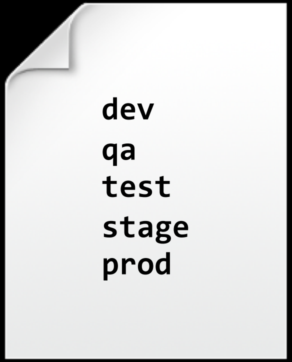
It’s tempting to define your entire infrastructure in 1 file / folder…



```
dev
qa
test
stage
prod
```

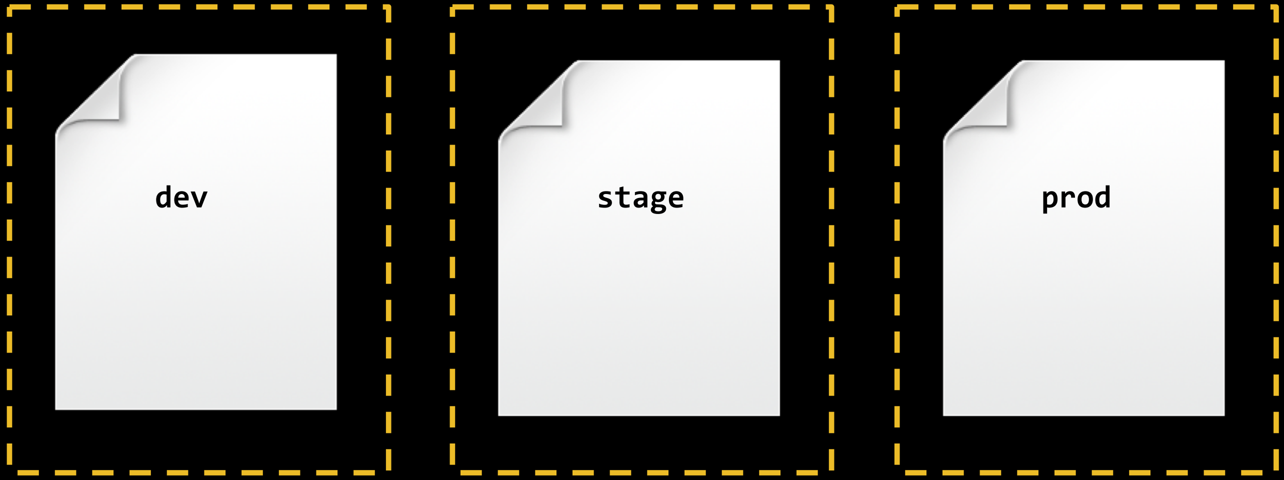
Downsides: runs slower; harder to understand; harder to review (`plan` output unreadable); harder to test; harder to reuse code; need admin permissions; team concurrency limited to 1...

Also, a mistake *anywhere* could break everything!

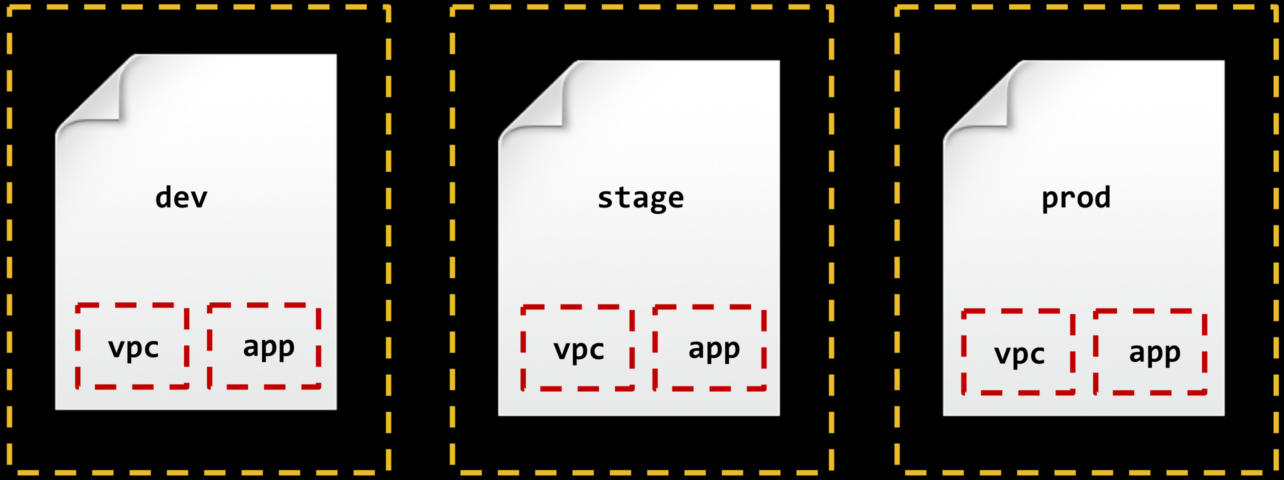


dev
qa
test
stage
prod

Large modules considered harmful.

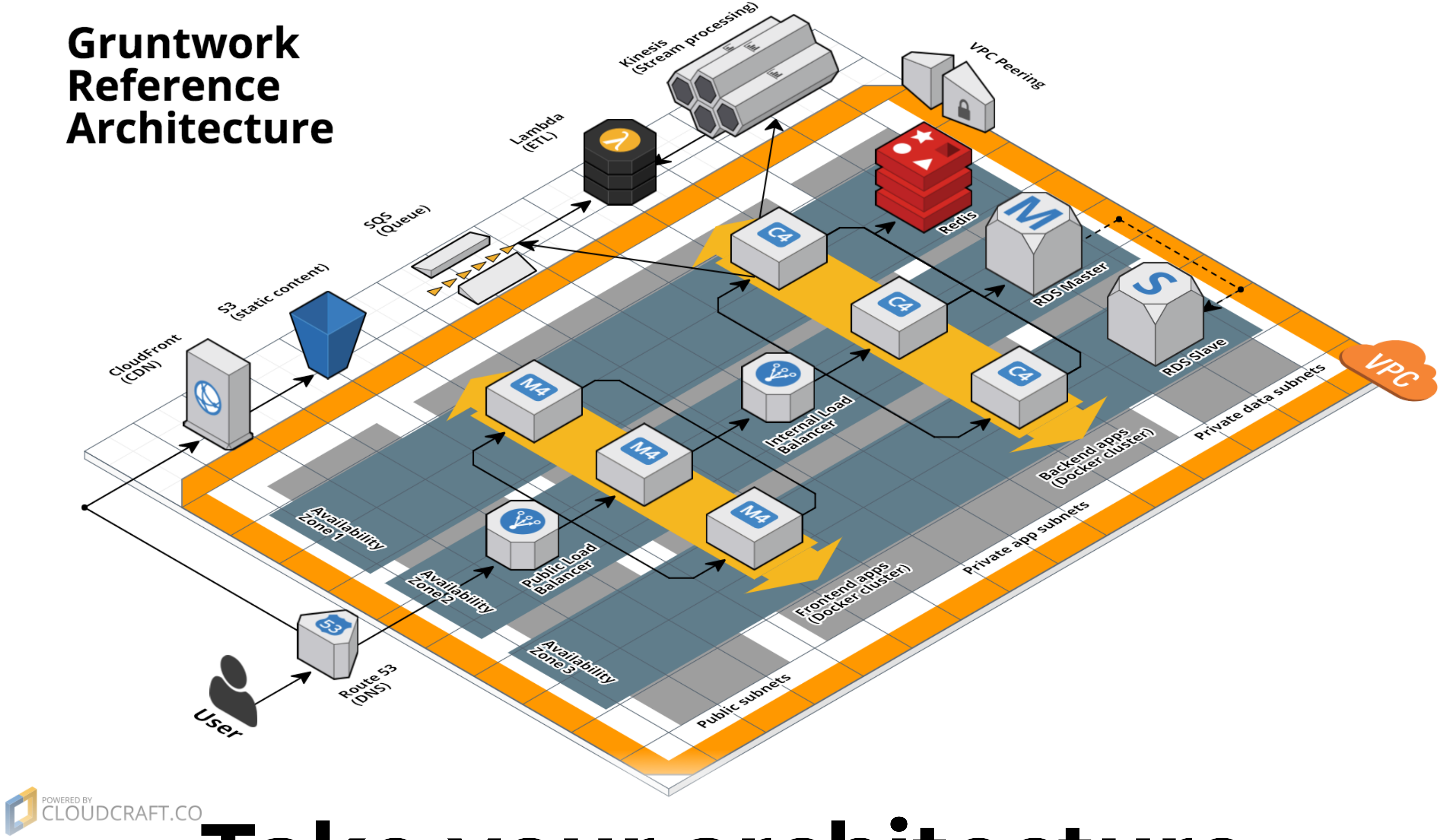


What you really want is **isolation** for each environment

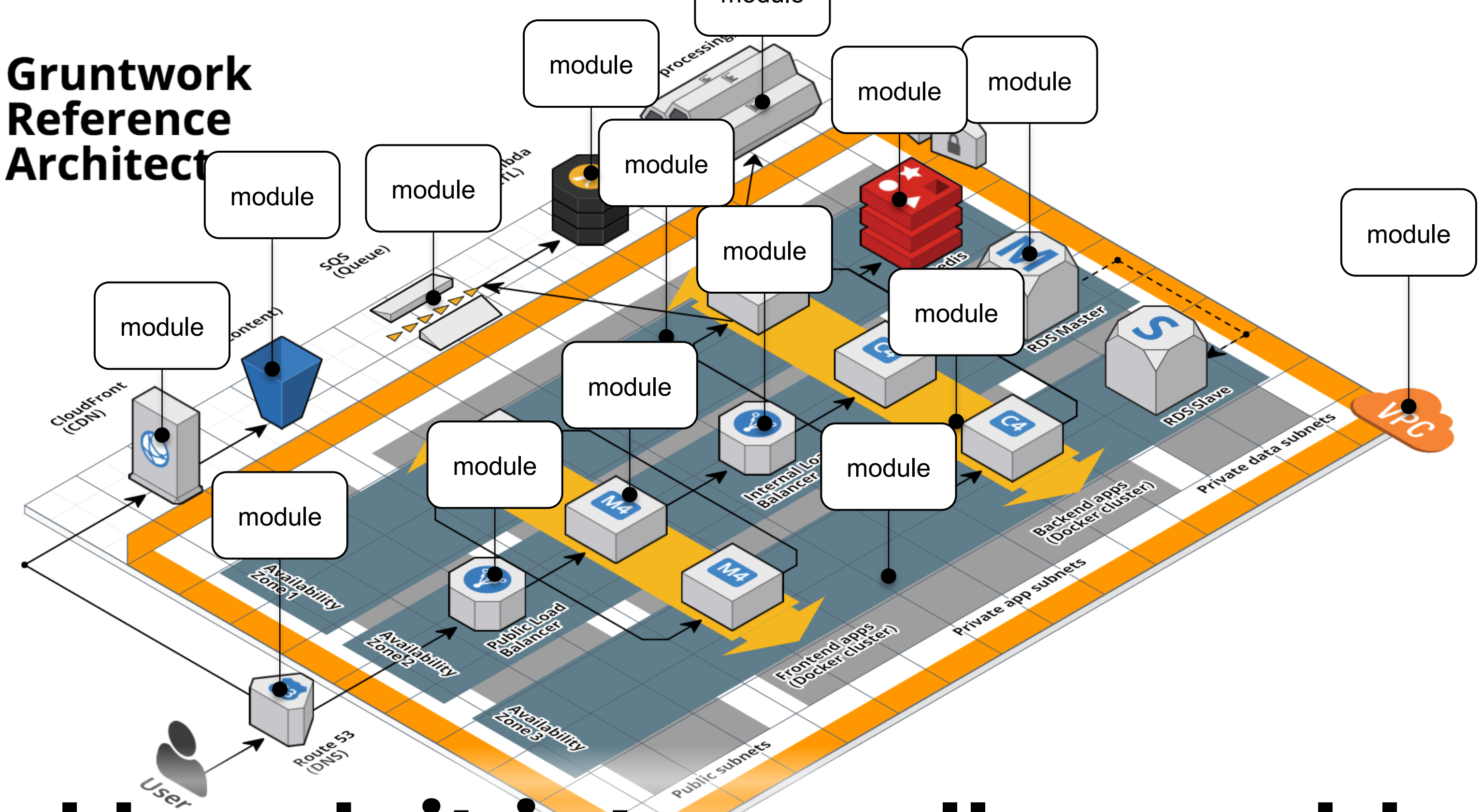


And for each “component”

Gruntwork Reference Architecture



Take your architecture...



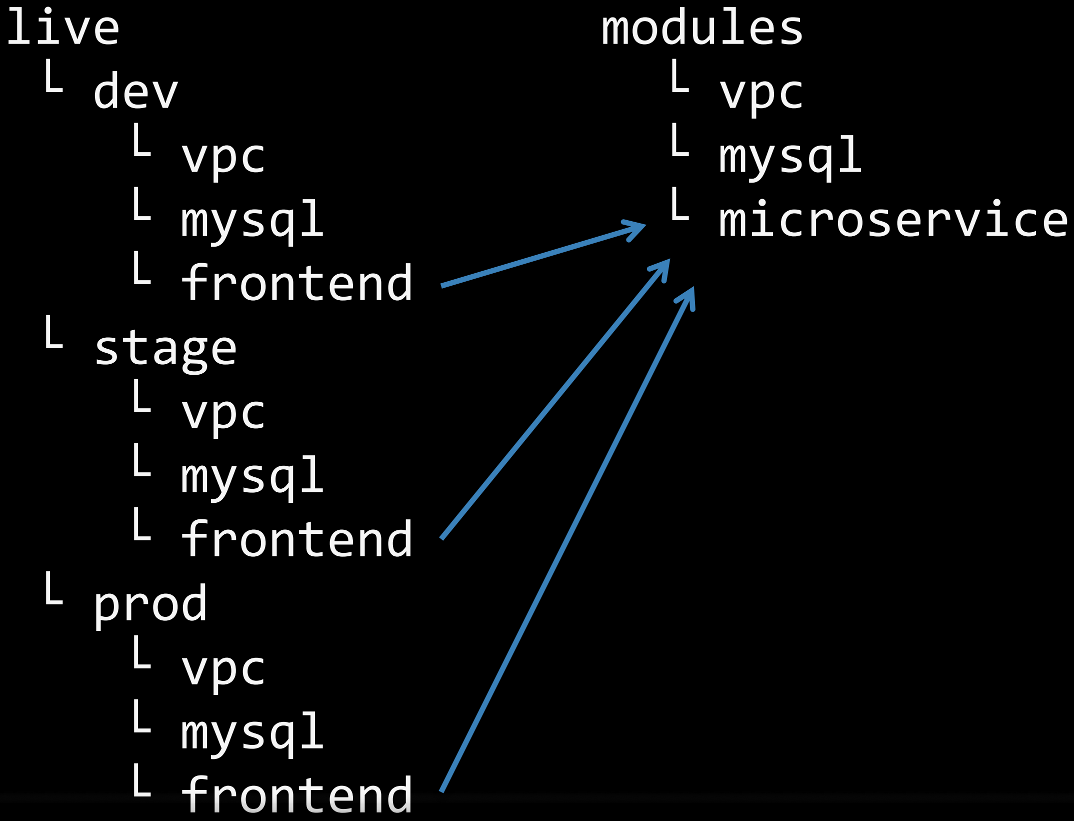
And break it into small, reusable, standalone, tested modules

```
live
 └ dev
 └ stage
 └ prod
```

Break architecture down by environment

```
live
 └ dev
    └ vpc
    └ mysql
    └ frontend
 └ stage
    └ vpc
    └ mysql
    └ frontend
 └ prod
    └ vpc
    └ mysql
    └ frontend
```

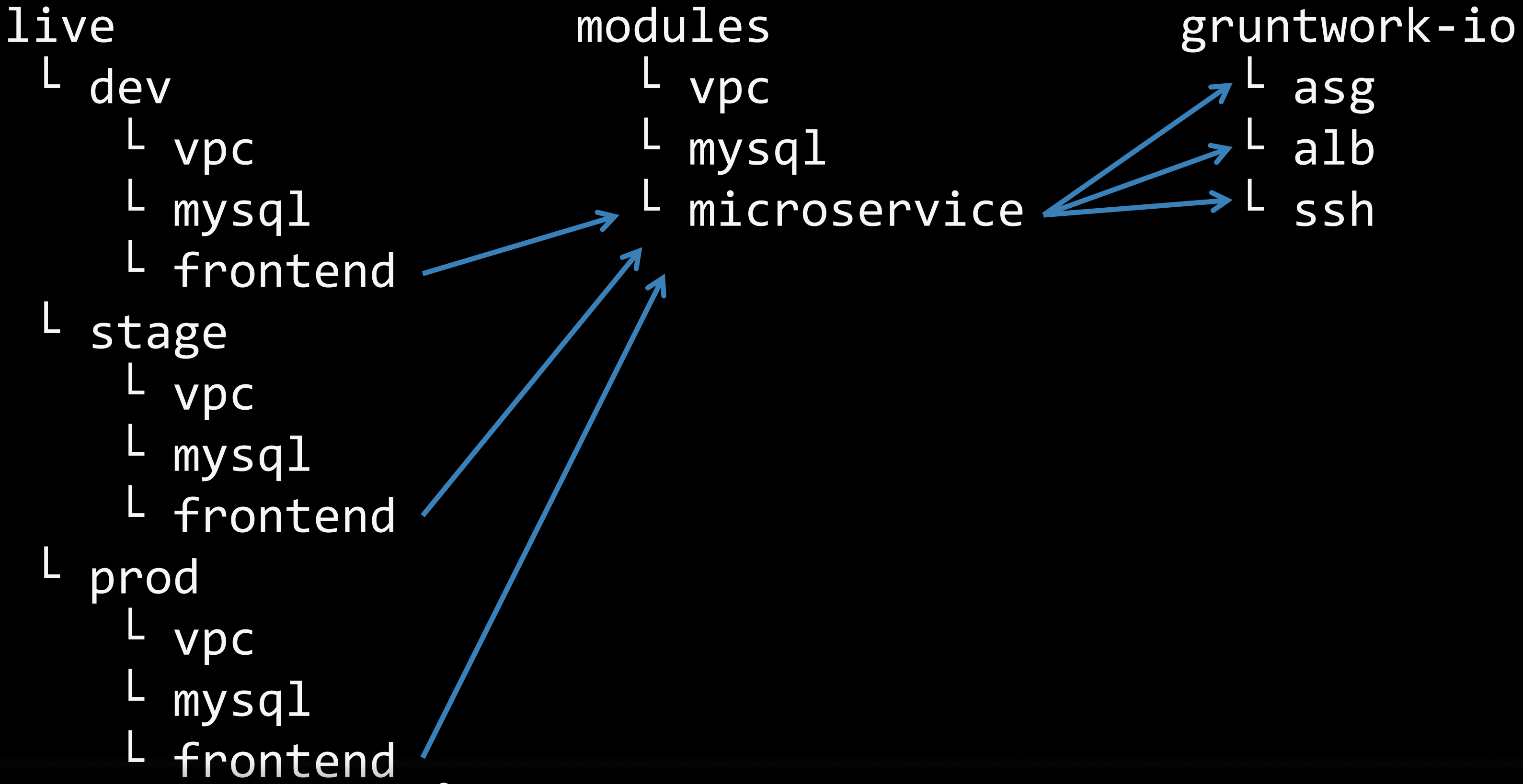
Break environments down by infrastructure type



```
live
 └ dev
    └ vpc
    └ mysql
    └ frontend
 └ stage
    └ vpc
    └ mysql
    └ frontend
 └ prod
    └ vpc
    └ mysql
    └ frontend
```

```
modules
 └ vpc
 └ mysql
 └ microservice
```

Implement infrastructure in modules

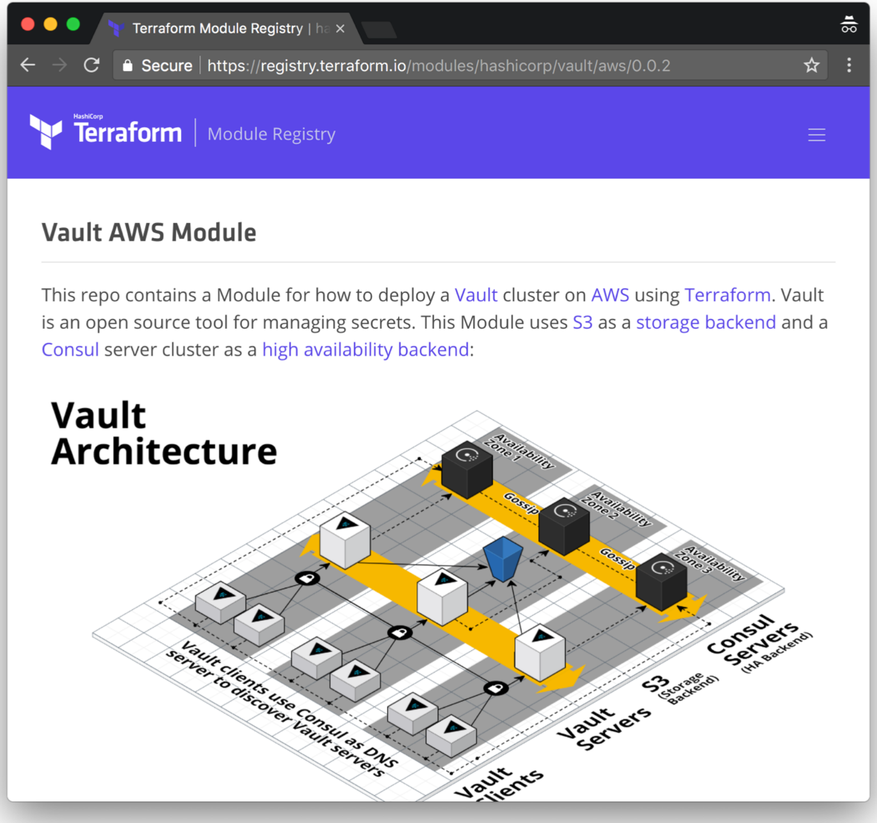


```
live
 └ dev
    └ vpc
    └ mysql
    └ frontend
 └ stage
    └ vpc
    └ mysql
    └ frontend
 └ prod
    └ vpc
    └ mysql
    └ frontend
```

```
modules
  └ vpc
  └ mysql
  └ microservice
```

```
gruntwork-io
  └ asg
  └ alb
  └ ssh
```

Build complex modules from simpler modules


Terraform Module Registry | ha
Secure
https://registry.terraform.io/modules/hashicorp/vault/aws/0.0.2
HashiCorp Terraform
Module Registry

Vault AWS Module

This repo contains a Module for how to deploy a Vault cluster on AWS using Terraform. Vault is an open source tool for managing secrets. This Module uses S3 as a storage backend and a Consul server cluster as a high availability backend:

Vault Architecture
Availability Zone 1
Availability Zone 2
Availability Zone 3
Gossip
Gossip
Vault clients use Consul as DNS server to discover Vault servers
Vault Clients
Vault Servers
S3 (Storage Backend)
Consul Servers (HA Backend)

Example: Vault Modules

```
terraform-aws-vault
  └ modules
  └ examples
  └ test
  └ README.md
```

Typical repo has three key folders:

`/modules`, `/examples`, `/test`

```
terraform-aws-vault
  └ modules
    └ install-vault
    └ run-vault
    └ vault-cluster
    └ vault-elb
    └ vault-security-group-rules
  └ examples
  └ test
  └ README.md
```

`/modules`: implementation code, broken down into standalone sub-modules

```
terraform-aws-vault
  └ modules
    └ install-vault
      └ install-vault.sh
    └ run-vault
    └ vault-cluster
    └ vault-security-group-rules
    └ vault-elb
  └ examples
  └ test
  └ README.md
```

`install-xxx`: sub-module to install the software (e.g., in Packer or Docker)

```
terraform-aws-vault
  └ modules
    └ install-vault
    └ run-vault
       └ run-vault.sh
    └ vault-cluster
    └ vault-security-group-rules
    └ vault-elb
  └ examples
  └ test
  └ README.md
```

`run-xxx`: sub-module to launch the software during boot (e.g., in User Data)

```
terraform-aws-vault
  └ modules
    └ install-vault
    └ run-vault
    └ vault-cluster
       └ main.tf
    └ vault-security-group-rules
    └ vault-elb
  └ examples
  └ test
  └ README.md
```

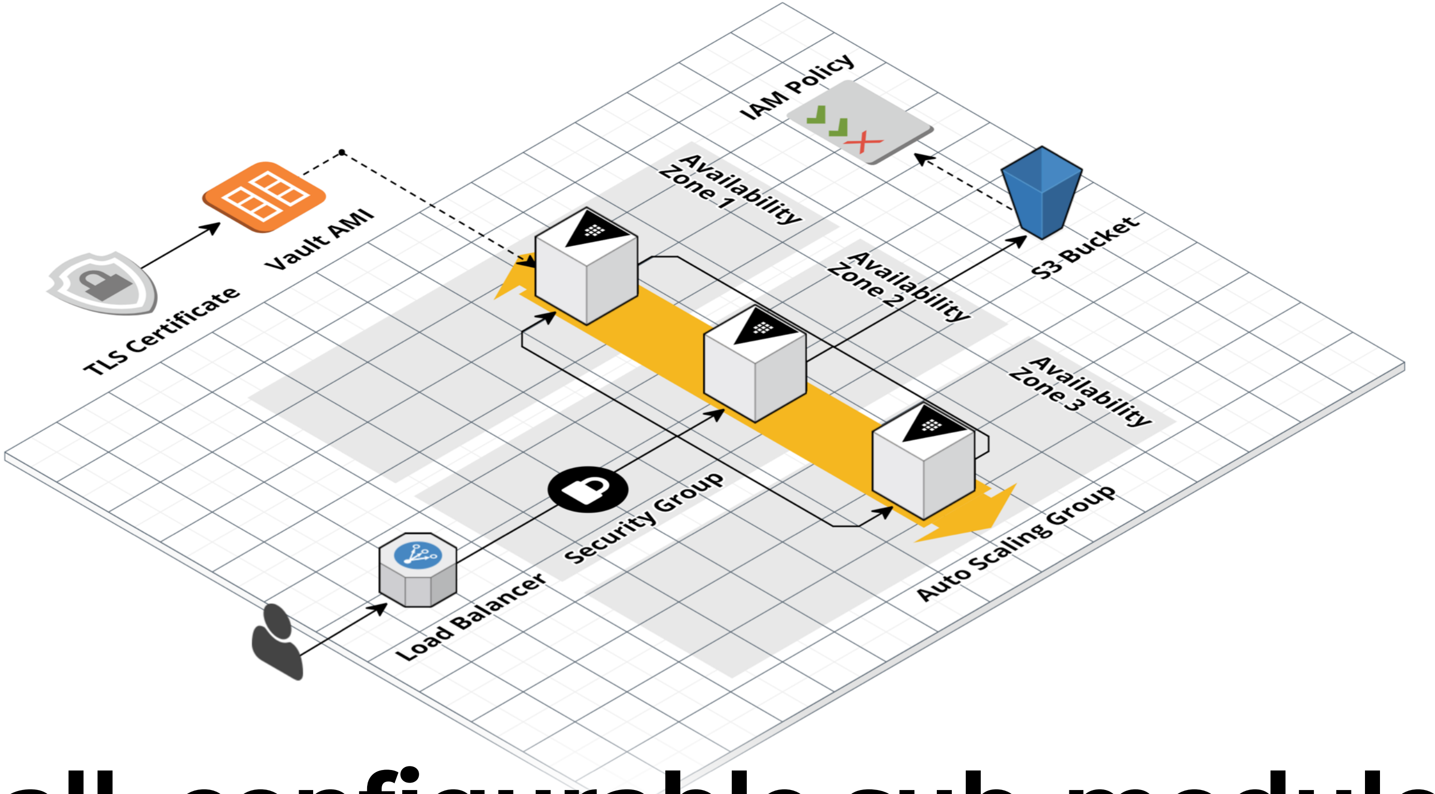
`xxx-cluster`: sub-module to deploy infrastructure (e.g., into an ASG)

```
terraform-aws-vault
  └ modules
    └ install-vault
    └ run-vault
    └ vault-cluster
    └ vault-security-group-rules
       └ main.tf
    └ vault-elb
       └ main.tf
  └ examples
  └ test
  └ README.md
```

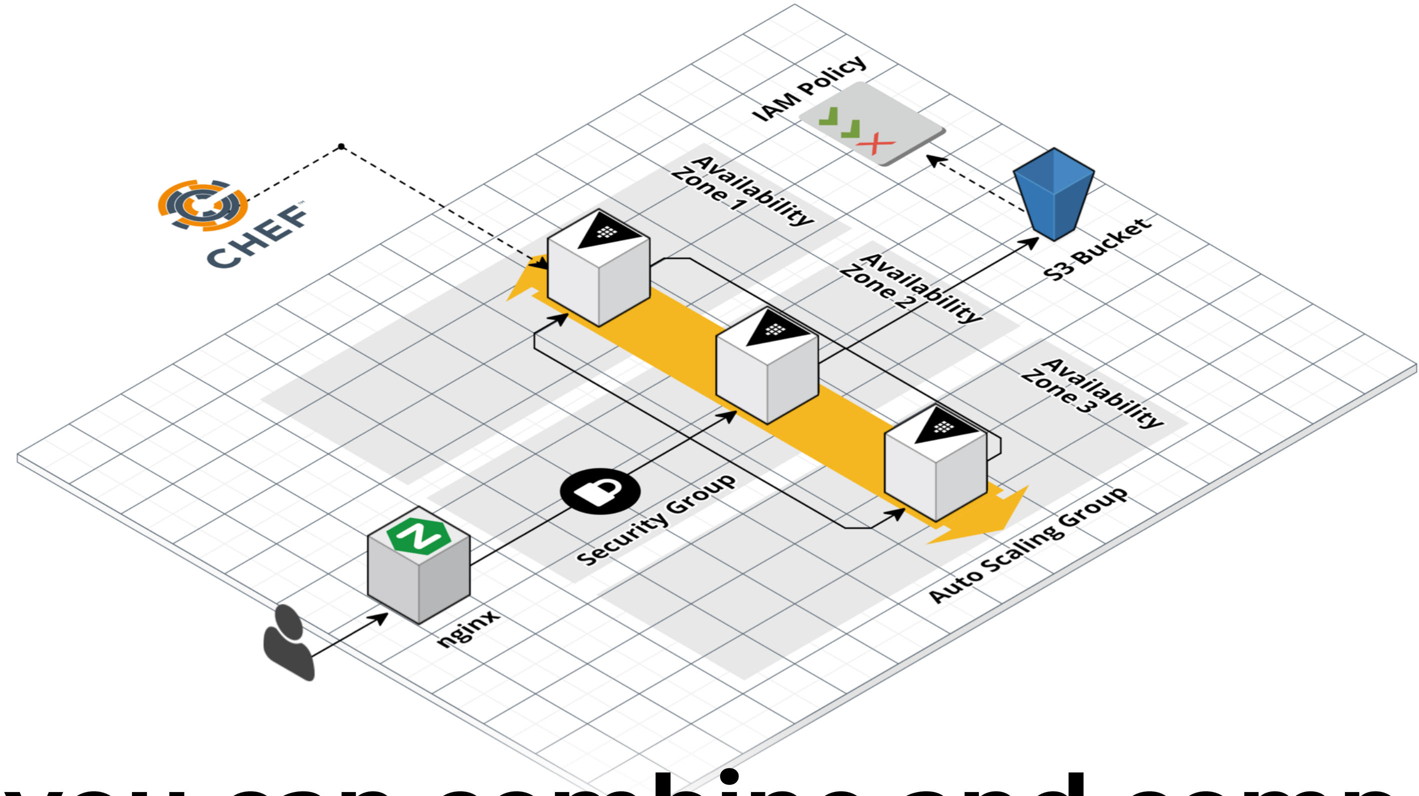
xxx-yyy: sub-modules with shareable components (e.g., Security Group rules)

```
variable "cluster_name" {
  description = "Name to use for the Vault cluster"
}

variable "vpc_id" {
  description = "ID of the VPC to use"
}

variable "allowed_inbound_cidr_blocks" {
  description = "IPs allowed to connect to Vault"
  type        = "list"
}
```

Each sub-module exposes variables for configuration and dependencies



Small, configurable sub-modules make code reuse possible



As you can combine and compose them any way you want!

```
terraforn-aws-vault
  └ modules
  └ examples
    └ vault-with-elb
    └ vault-s3-backend
    └ vault-ami
  └ test
  └ README.md
```

`/examples`: **Runnable example code for how to use the sub-modules**

```
terraform-aws-vault
  └ modules
  └ examples
    └ vault-with-elb
      └ main.tf
    └ vault-s3-backend
      └ main.tf
    └ vault-ami
  └ test
  └ README.md
```

Some examples are Terraform code

```
terraform-aws-vault
  └ modules
  └ examples
    └ vault-with-elb
    └ vault-s3-backend
    └ vault-ami
       └ vault.json
  └ test
  └ README.md
```

Some examples are Packer templates or Dockerfiles

```
terraform-aws-vault
  └ modules
  └ examples
  └ test
     └ vault_with_elb_test.go
     └ vault_s3_backend_test.go
  └ README.md
```

/tests: Automated tests for the sub-modules.

```
terraform-aws-vault
  └ modules
  └ examples
    └ vault-with-elb
    └ vault-s3-backend
  └ test
     └ vault_with_elb_test.go
     └ vault_s3_backend_test.go
  └ README.md
```

Typically, our tests deploy & validate each example! More on this later.

Key takeaway: build infrastructure from small, composable modules.

Outline

1. Checklist
2. Tools
3. Modules
4. **Tests**
5. Releases

Infrastructure code rots very quickly.

~~Infrastructure code rots very quickly.~~

Infrastructure code without automated tests is broken.

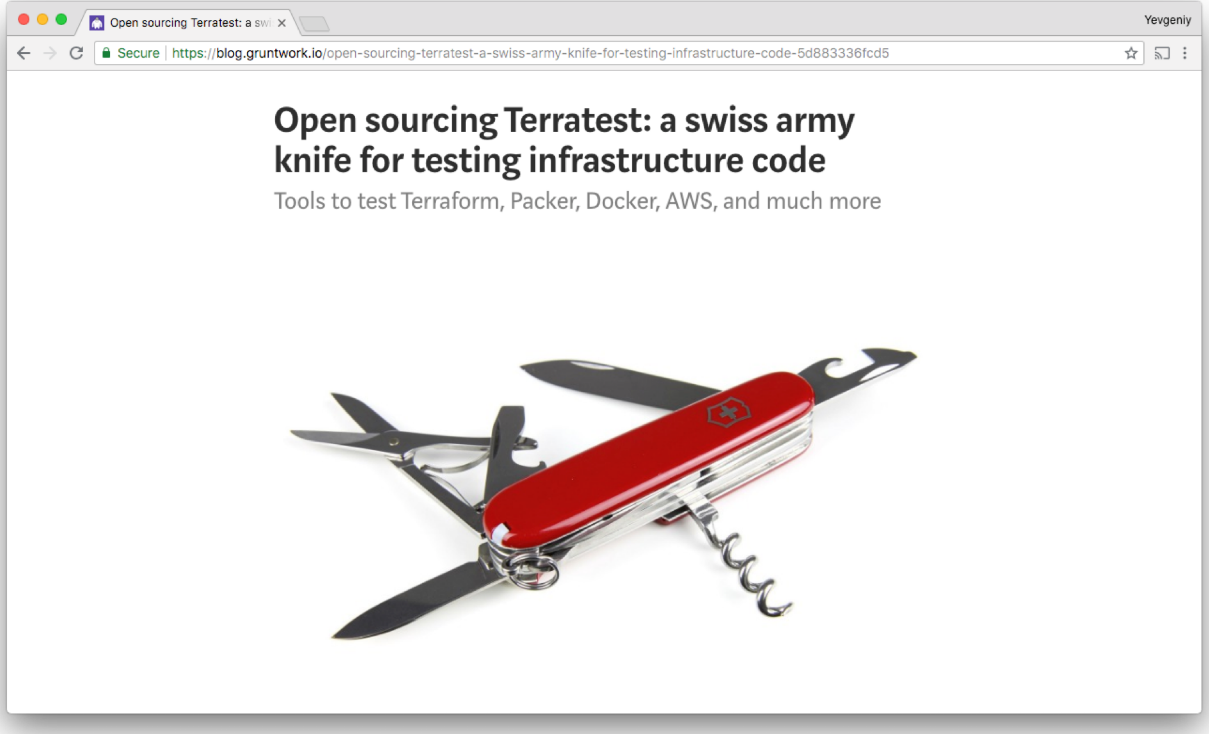
For general-purpose languages, we can run unit tests on localhost

Terraform

For infrastructure as code tools, there is no “localhost” or “unit”

Therefore, the test strategy is:

1. Deploy real infrastructure
2. Validate it works
3. Undeploy the infrastructure


Open sourcing Terratest: a swi
Yevgeniy
Secure
https://blog.gruntwork.io/open-sourcing-terratest-a-swiss-army-knife-for-testing-infrastructure-code-5d883336fcd5

Open sourcing Terratest: a swiss army knife for testing infrastructure code

Tools to test Terraform, Packer, Docker, AWS, and much more

We write these integration tests in Go using Terratest (open source!)

Terratest philosophy: how would you test it manually?

```
terraformOptions := &terraform.Options {
  TerraformDir: "../examples/vault-with-elb",
}

defer terraform.Destroy(t, terraformOptions)

terraform.InitAndApply(t, terraformOptions)

validateServerIsWorking(t, terraformOptions)
```

Typical test structure

```
terraformOptions := &terraform.Options {
  TerraformDir: "../examples/vault-with-elb",
}

defer terraform.Destroy(t, terraformOptions)

terraform.InitAndApply(t, terraformOptions)

validateServerIsWorking(t, terraformOptions)
```

Specify where the code lives

```
terraformOptions := &terraform.Options {
  TerraformDir: "../examples/vault-with-elb",
}

defer terraform.Destroy(t, terraformOptions)

terraform.InitAndApply(t, terraformOptions)

validateServerIsWorking(t, terraformOptions)
```

Run `terraform init` **and** `terraform apply` **to deploy**

```
terraformOptions := &terraform.Options {
  TerraformDir: "../examples/vault-with-elb",
}

defer terraform.Destroy(t, terraformOptions)

terraform.InitAndApply(t, terraformOptions)

validateServerIsWorking(t, terraformOptions)
```

Validate the infrastructure works as expected

```
// Get IPs of servers
aws.GetPublicIpsOfEc2Instances(t, ids, region)

// Make HTTP requests in a retry loop
http.GetWithRetry(t, url, 200, expected, retries, sleep)

// Run command over SSH
ssh.CheckSshCommand(t, host, "vault operator init")
```

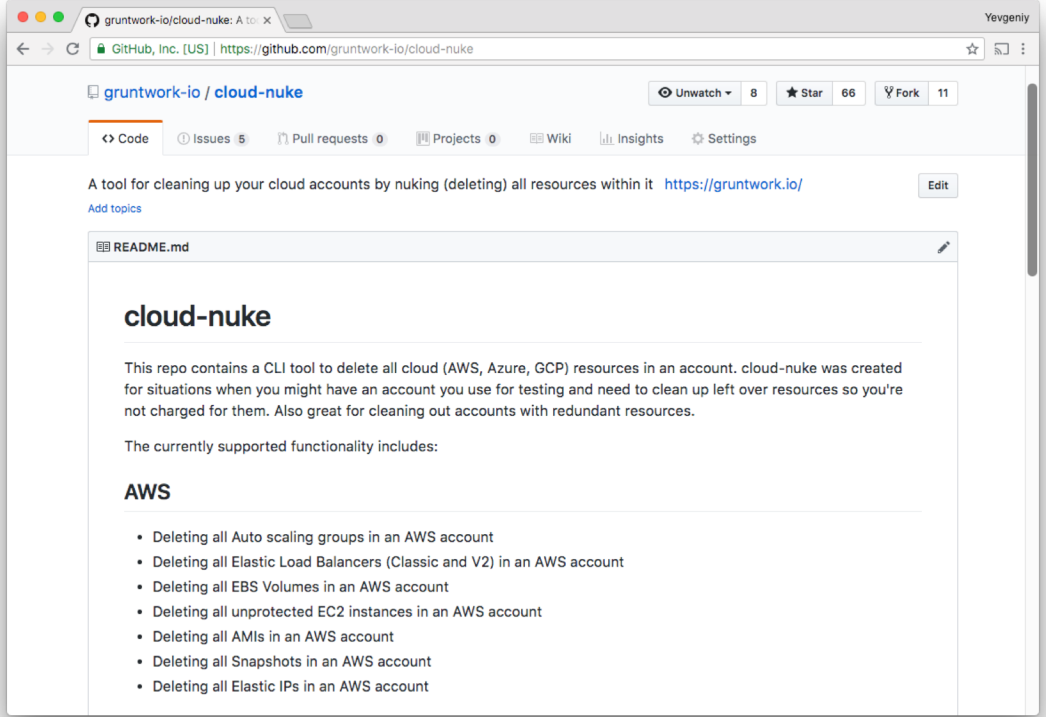
Terratest has many tools built-in for validation

```
terraformOptions := &terraform.Options {
  TerraformDir: "../examples/vault-with-elb",
}

defer terraform.Destroy(t, terraformOptions)

terraform.InitAndApply(t, terraformOptions)

validateServerIsWorking(t, terraformOptions)
```

At the end of the test, run `terraform destroy` to clean up

Note: tests create and destroy lots of resources!

Pro tip #1: run tests in completely separate “sandbox” accounts


gruntwork-io/cloud-nuke: A to
Yevgeniy
GitHub, Inc. [US] | https://github.com/gruntwork-io/cloud-nuke
gruntwork-io / cloud-nuke
Unwatch 8
Star 66
Fork 11
Code
Issues 5
Pull requests 0
Projects 0
Wiki
Insights
Settings
Edit
Add topics
README.md

A tool for cleaning up your cloud accounts by nuking (deleting) all resources within it https://gruntwork.io/

cloud-nuke

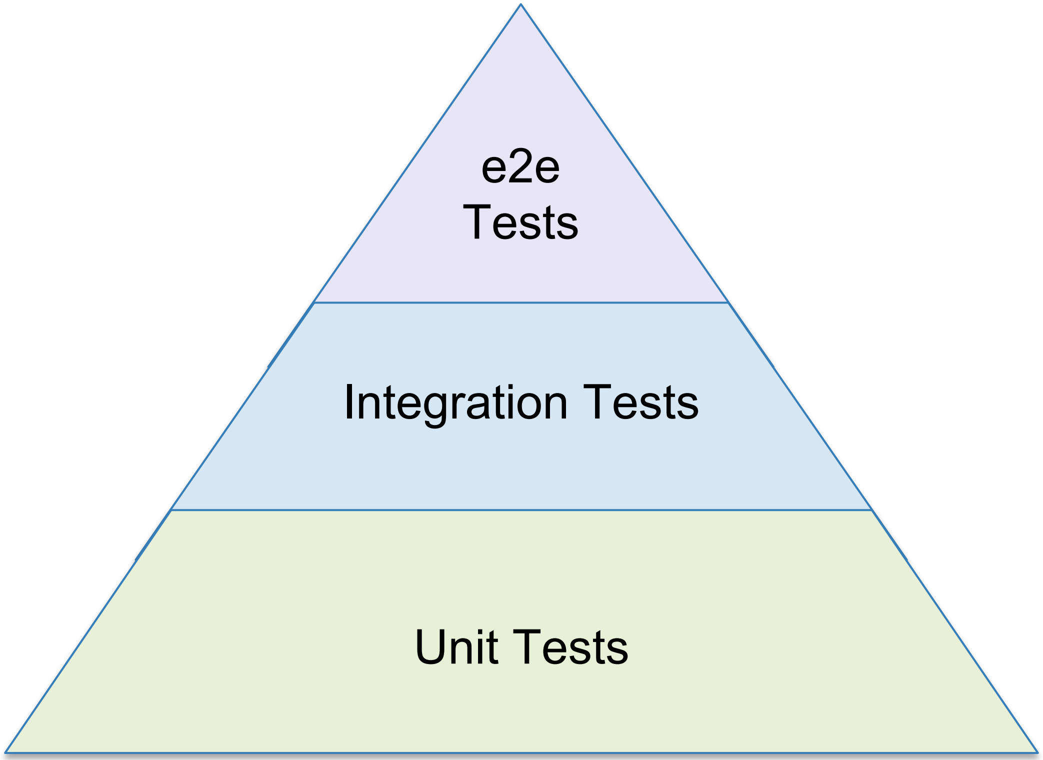
This repo contains a CLI tool to delete all cloud (AWS, Azure, GCP) resources in an account. cloud-nuke was created for situations when you might have an account you use for testing and need to clean up left over resources so you're not charged for them. Also great for cleaning out accounts with redundant resources.

The currently supported functionality includes:

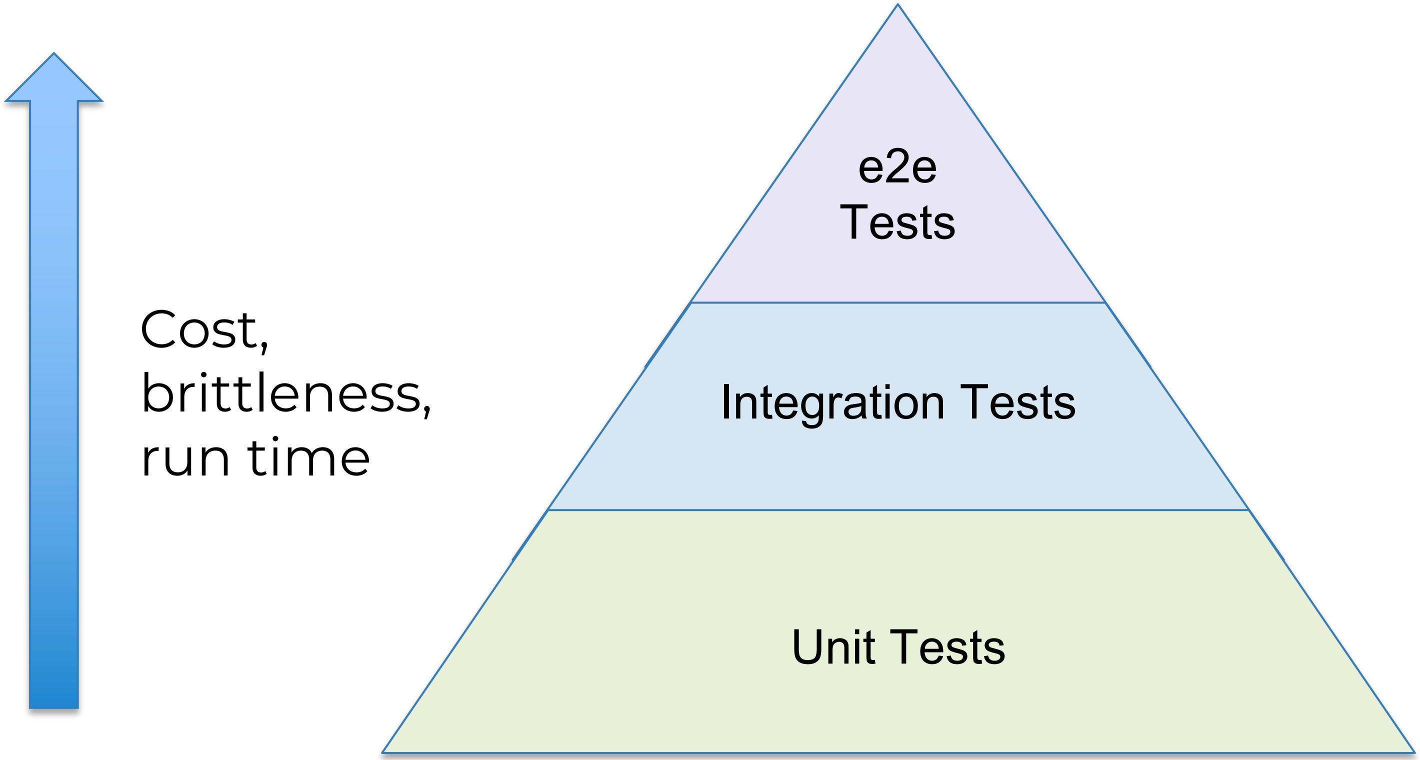
AWS

- Deleting all Auto scaling groups in an AWS account
- Deleting all Elastic Load Balancers (Classic and V2) in an AWS account
- Deleting all EBS Volumes in an AWS account
- Deleting all unprotected EC2 instances in an AWS account
- Deleting all AMIs in an AWS account
- Deleting all Snapshots in an AWS account
- Deleting all Elastic IPs in an AWS account

Pro tip #2: clean up left-over resources with cloud-nuke.

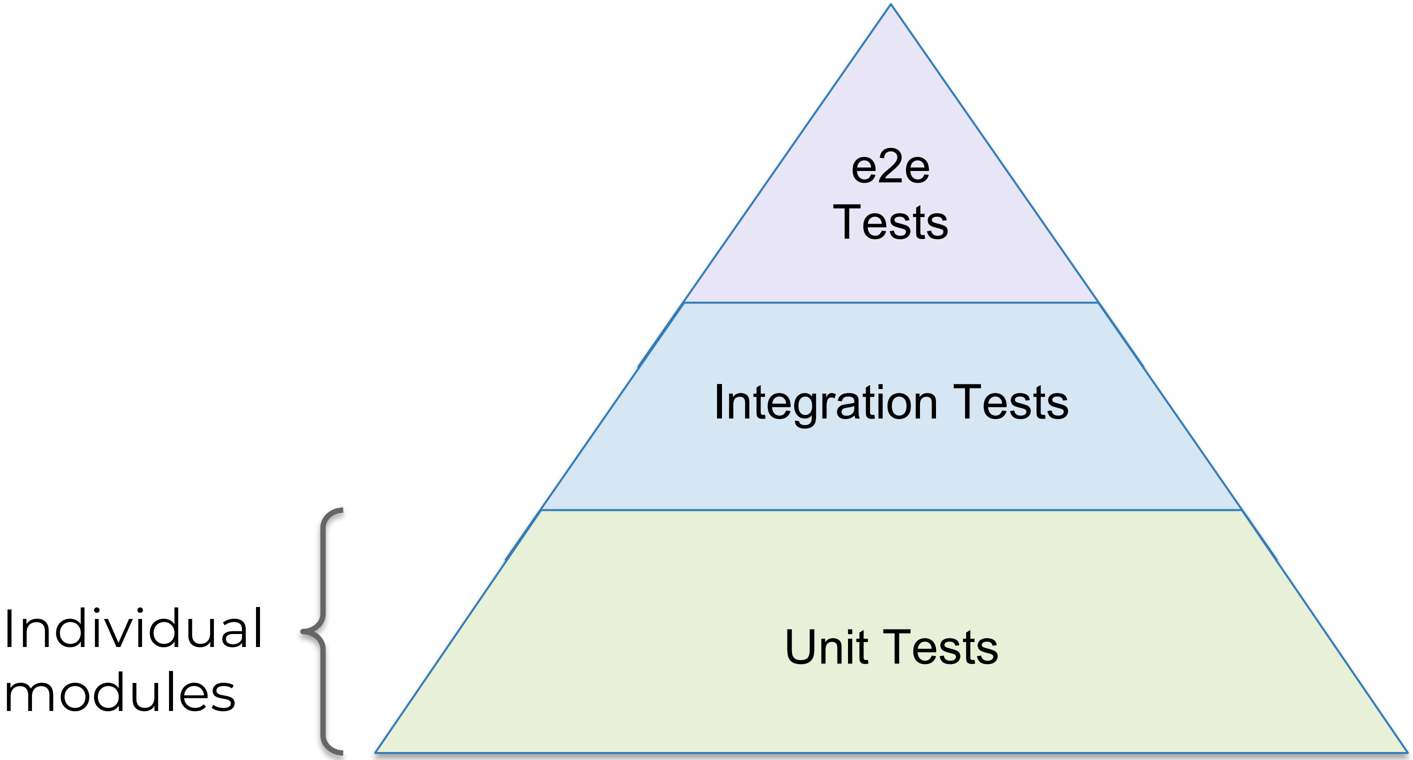


Test pyramid

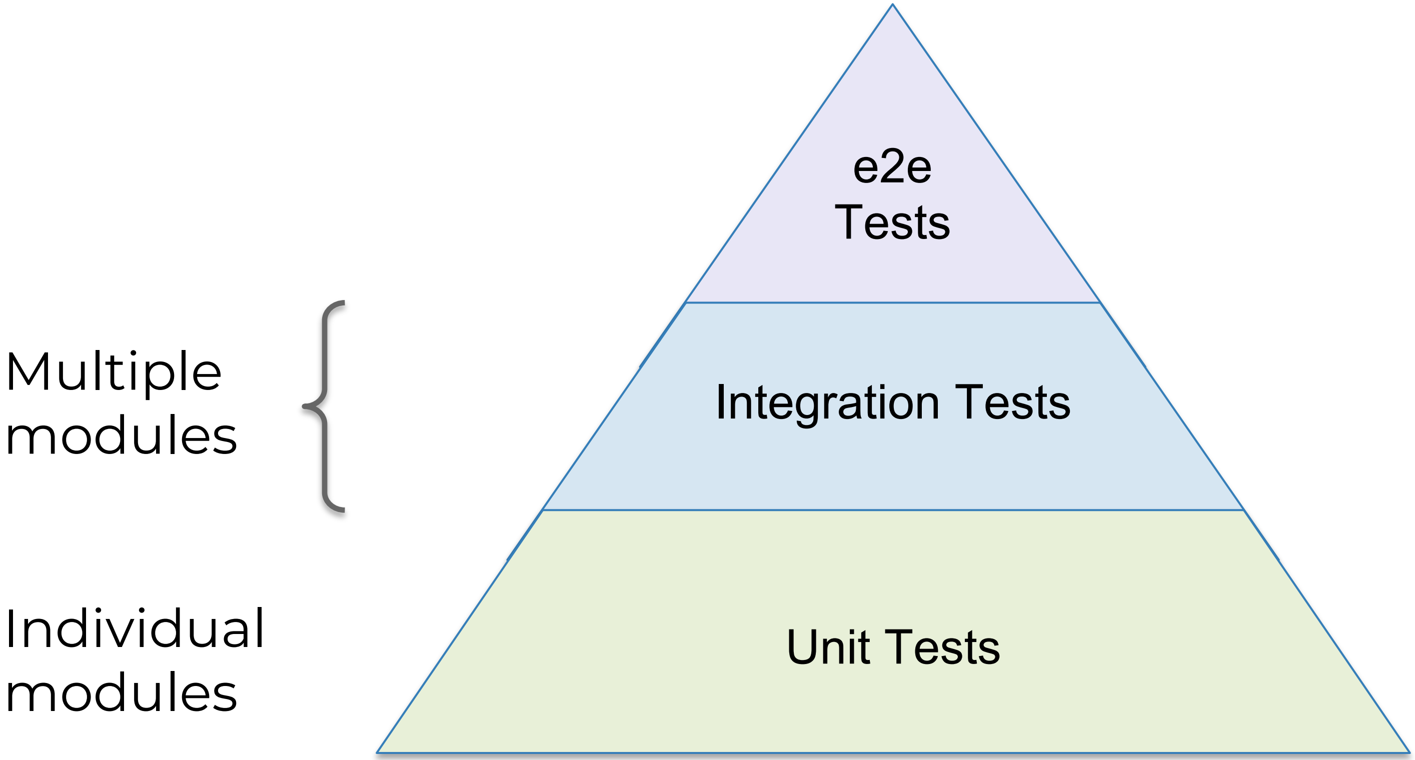


As you go up the pyramid, tests get more expensive, brittle, and slower

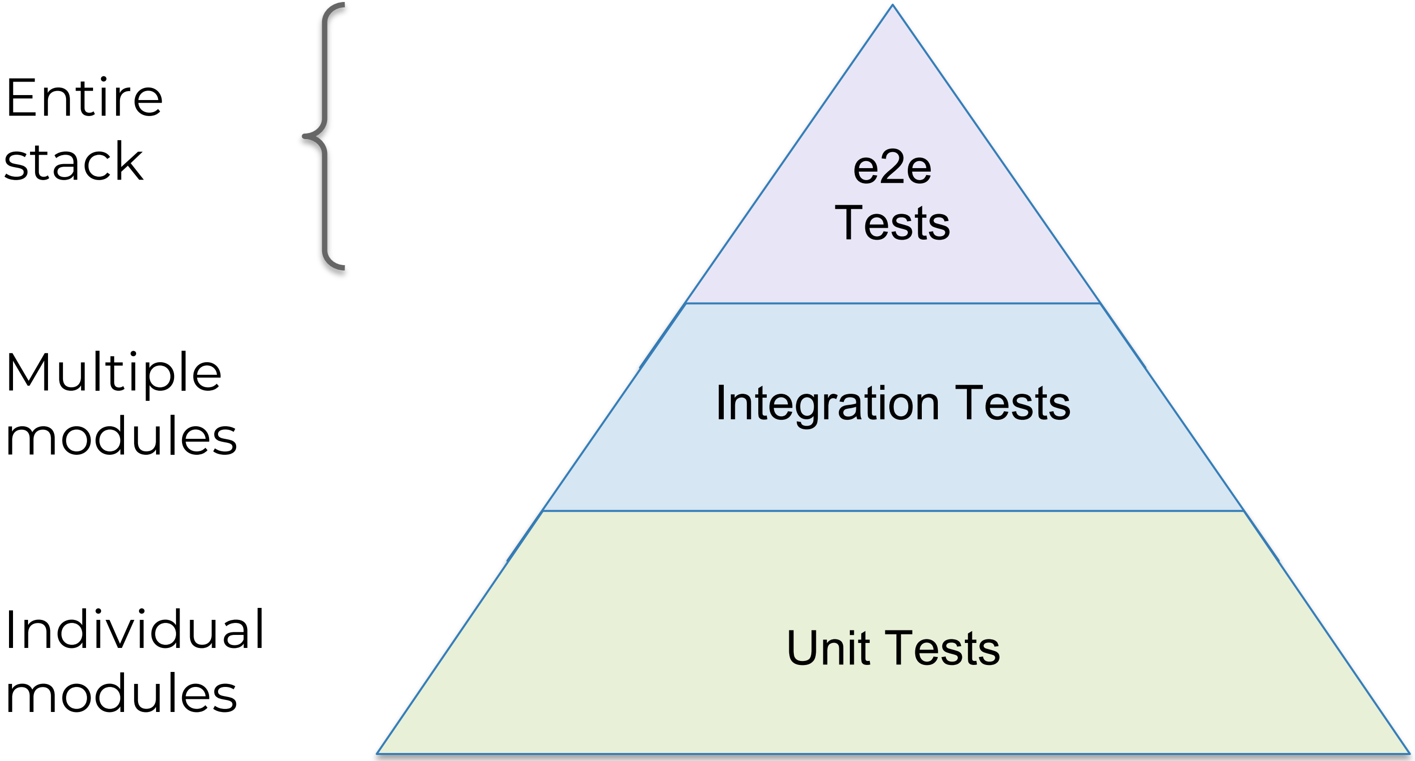
How the test pyramid works with infrastructure code:



Unit tests for infrastructure: test individual sub-modules (keep 'em small!)



Integration tests for infrastructure: test multiple sub-modules together.

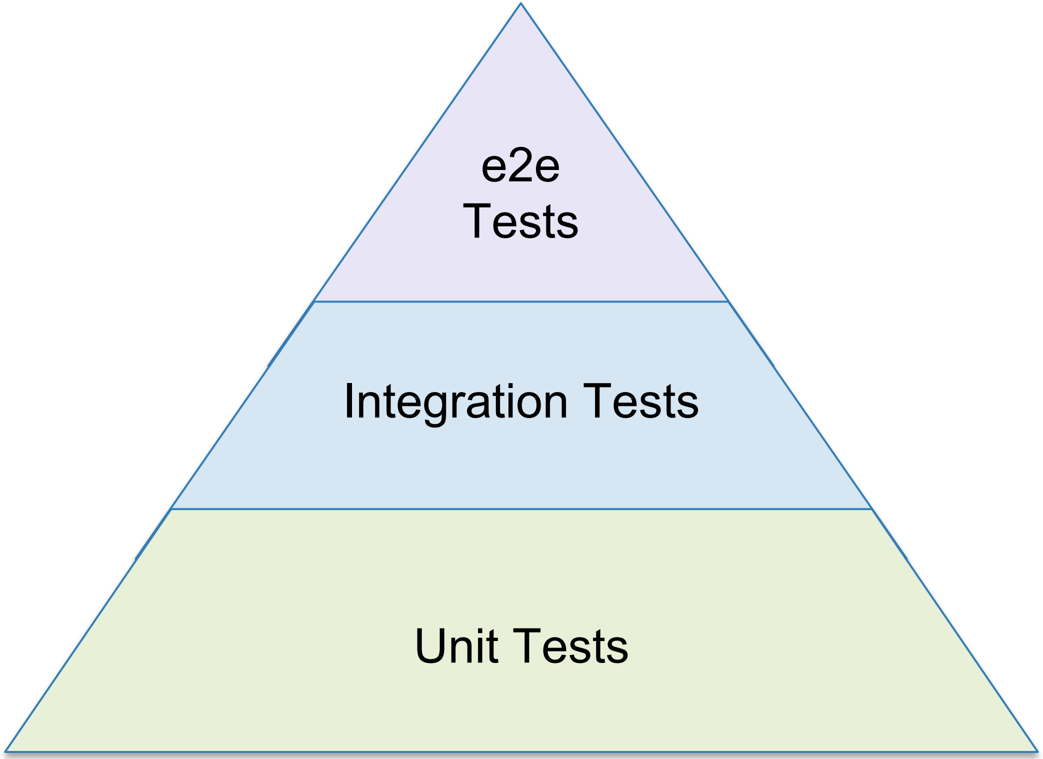


E2E tests for infrastructure code: test entire environments (stage, prod).

30 – 120+ minutes

5 – 60 minutes

1 – 20 minutes



Note the test times! This is another reason to use small modules.

Key takeaway: infrastructure code without automated tests is broken.

Outline

1. Checklist
2. Tools
3. Modules
4. Tests
5. Releases

Let’s put it all together: checklist, tools, modules, tests

Task	Description	Example tools
Security	Encryption in transit (TLS) and on disk, authentication, authorization, secrets management, server hardening.	ACM, EBS Volumes, Cognito, Vault, CiS
Monitoring	Availability metrics, business metrics, app metrics, server, metrics, events, observability, tracing, alerting.	CloudWatch, DataDog, New Relic, Honeycomb
Logs	Rotate logs on disk. Aggregate log data to a central location.	CloudWatch Logs, ELK, Sumo Logic, Papertrail
Backup and restore	Make backups of DBs, caches, and other data on a scheduled basis. Replicate to separate region/account.	RDS, ElastiCache, ec2-snapper, Lambda

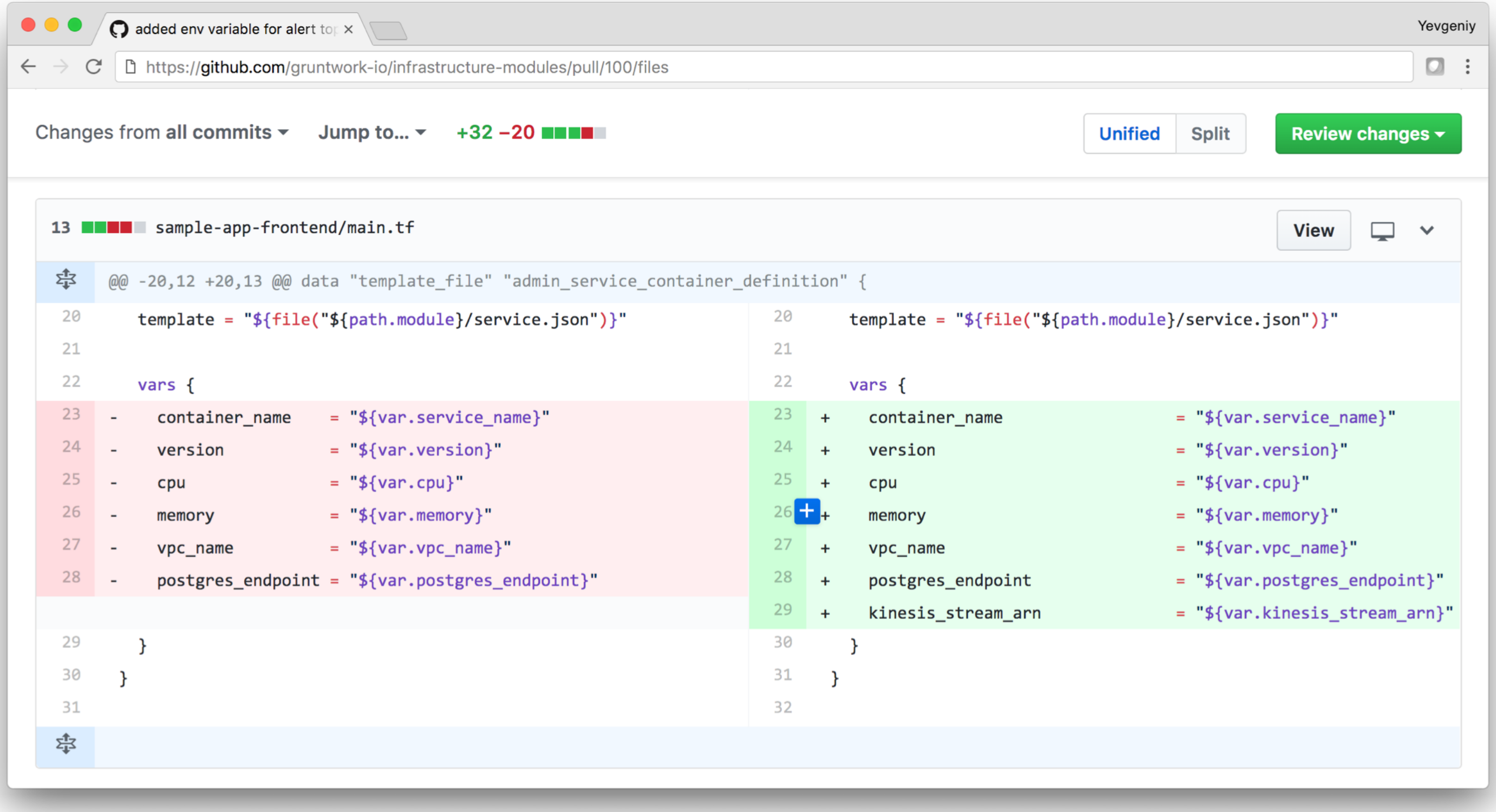
1. Go through the checklist

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

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

2. Write some code

```
terraformOptions := &terraform.Options {
  TerraformDir: "../examples/vault-with-elb",
}

defer terraform.Destroy(t, terraformOptions)

terraform.InitAndApply(t, terraformOptions)

validateServerIsWorking(t, terraformOptions)
```

3. Write automated tests

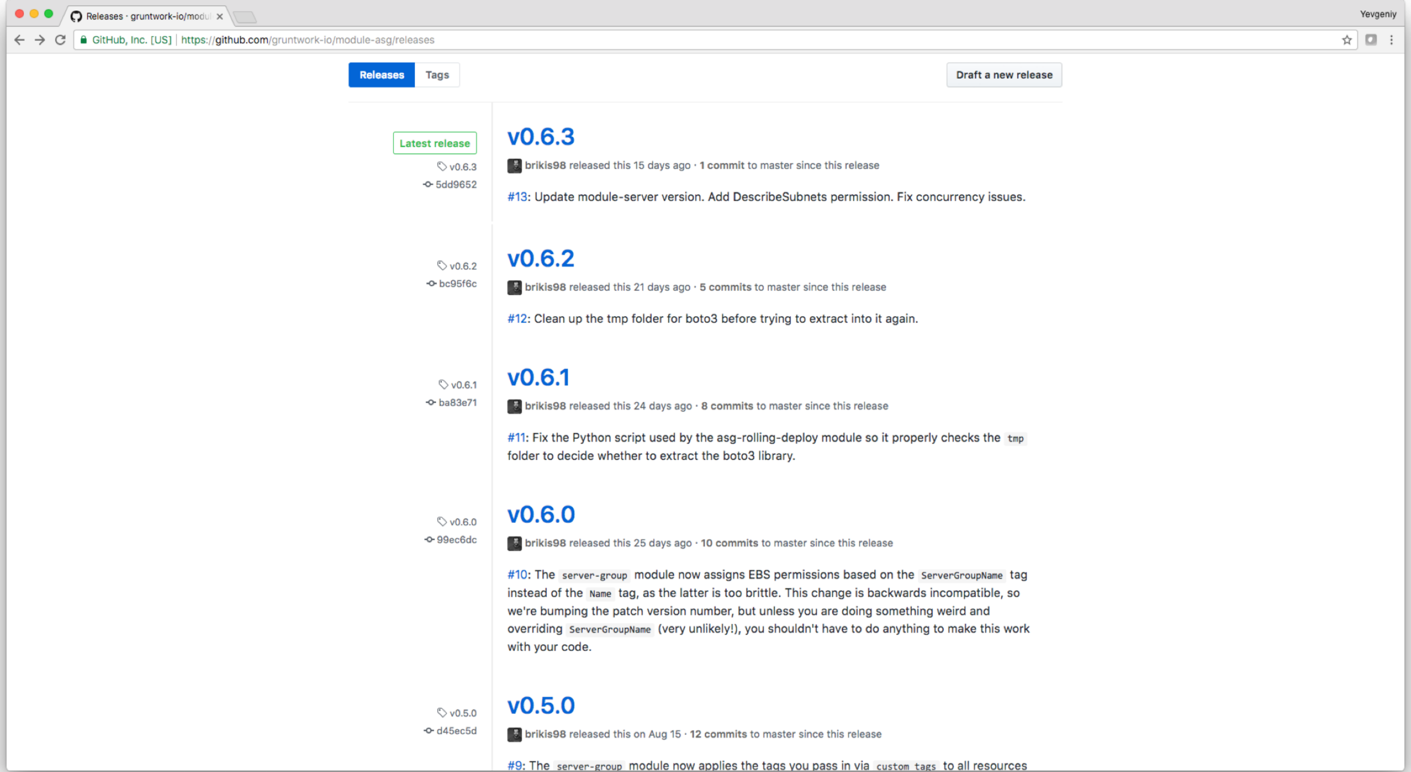

added env variable for alert to
Yevgeniy
https://github.com/gruntwork-io/infrastructure-modules/pull/100/files
Changes from all commits
Jump to...
+32 −20
Unified
Split
Review changes
13
sample-app-frontend/main.tf
View

```
@@ -20,12 +20,13 @@ data "template_file" "admin_service_container_definition" {
```

```
  template = "${file("${path.module}/service.json")}"

  vars {
-   container_name    = "${var.service_name}"
-   version           = "${var.version}"
-   cpu               = "${var.cpu}"
-   memory            = "${var.memory}"
-   vpc_name          = "${var.vpc_name}"
-   postgres_endpoint = "${var.postgres_endpoint}"
  }
}
```

```
  template = "${file("${path.module}/service.json")}"

  vars {
+   container_name                  = "${var.service_name}"
+   version                         = "${var.version}"
+   cpu                             = "${var.cpu}"
+   memory                          = "${var.memory}"
+   vpc_name                        = "${var.vpc_name}"
+   postgres_endpoint               = "${var.postgres_endpoint}"
+   kinesis_stream_arn              = "${var.kinesis_stream_arn}"
  }
}
```

4. Do a code review


Releases · gruntwork-io/modul
Yevgeniy
GitHub, Inc. [US] | https://github.com/gruntwork-io/module-asg/releases
Releases
Tags
Draft a new release
Latest release

v0.6.3
5dd9652

v0.6.3

brikis98 released this 15 days ago · 1 commit to master since this release

#13: Update module-server version. Add DescribeSubnets permission. Fix concurrency issues.

v0.6.2
bc95f6c

v0.6.2

brikis98 released this 21 days ago · 5 commits to master since this release

#12: Clean up the tmp folder for boto3 before trying to extract into it again.

v0.6.1
ba83e71

v0.6.1

brikis98 released this 24 days ago · 8 commits to master since this release

#11: Fix the Python script used by the asg-rolling-deploy module so it properly checks the `tmp` folder to decide whether to extract the boto3 library.

v0.6.0
99ec6dc

v0.6.0

brikis98 released this 25 days ago · 10 commits to master since this release

#10: The `server-group` module now assigns EBS permissions based on the `ServerGroupName` tag instead of the `Name` tag, as the latter is too brittle. This change is backwards incompatible, so we're bumping the patch version number, but unless you are doing something weird and overriding `ServerGroupName` (very unlikely!), you shouldn't have to do anything to make this work with your code.

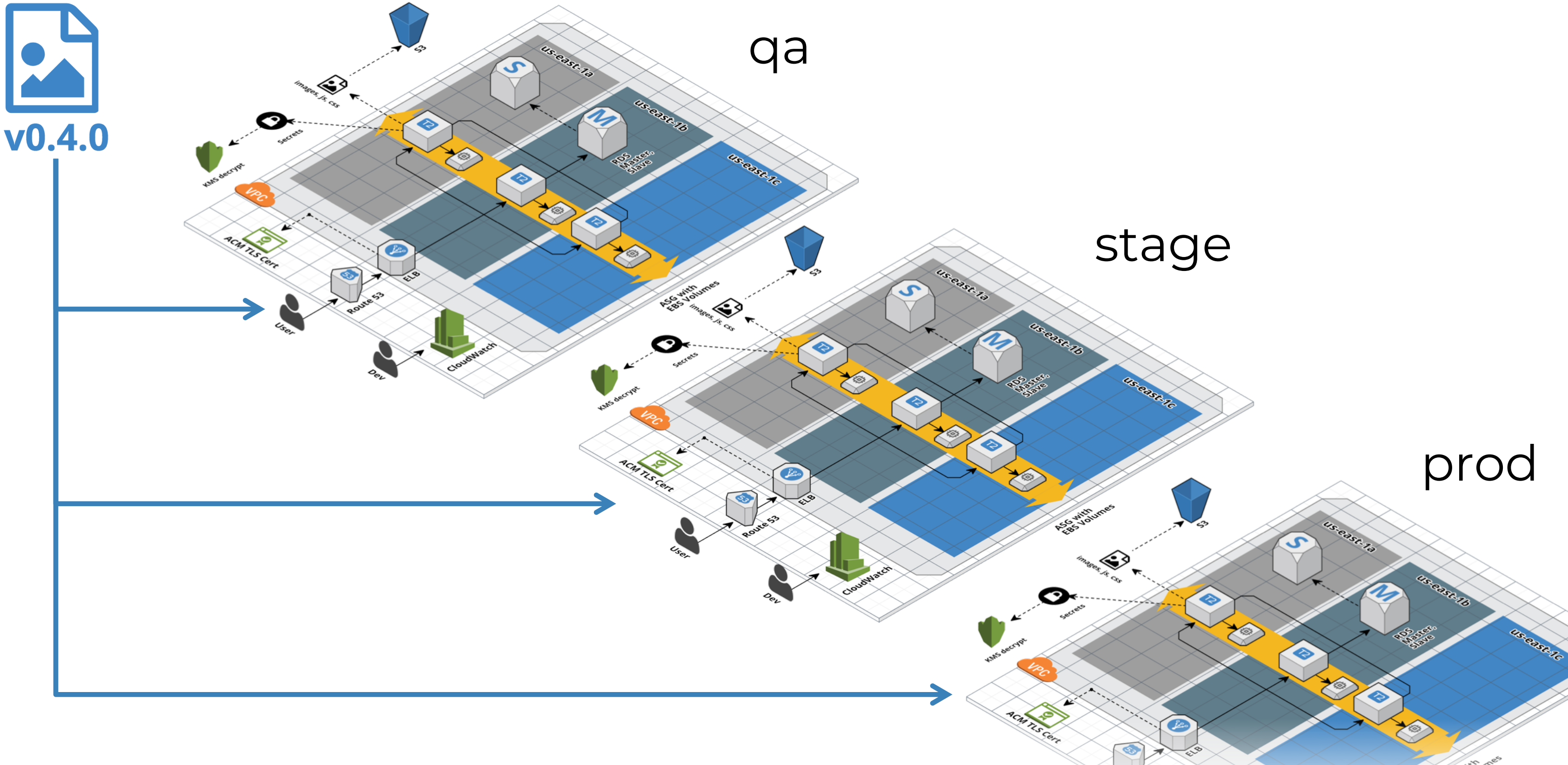
v0.5.0
d45ec5d

v0.5.0

brikis98 released this on Aug 15 · 12 commits to master since this release

#9: The `server-group` module now applies the tags you pass in via `custom_tags` to all resources

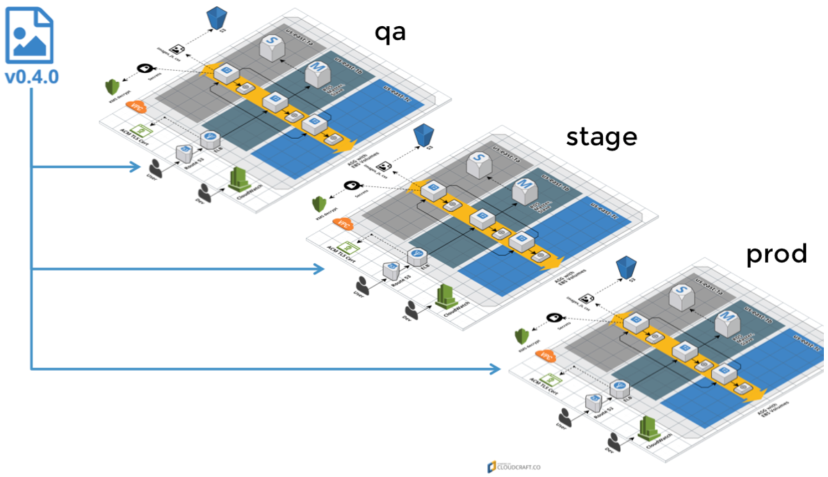
5. Release a new version of your code



6. Promote that versioned code from environment to environment

Key takeaway:

Before...



...After

Questions?

info@gruntwork.io