Nicki Watt

Evolving Your Infrastructure with Terraform

OpenCredo

#hashidays

BY HashiCorp

OpenCredo

ABOUT ME / OPENCREDO

- OpenCredo CTO
- Premiere HashiCorp partner
- Hands on software development consultancy
- Cloud, Data Engineering, DevSecOps

AGENDA

- Evolving your Terraform
- Orchestrating your Terraform
- Conclusion

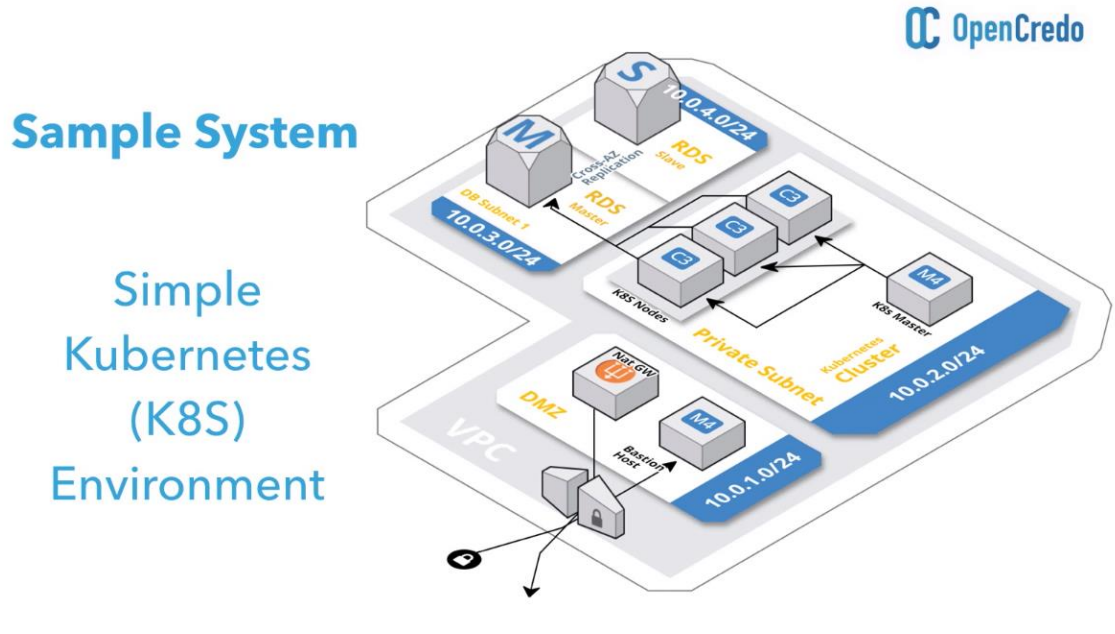
Evolving your Terraform

(a journey from a client's perspective)

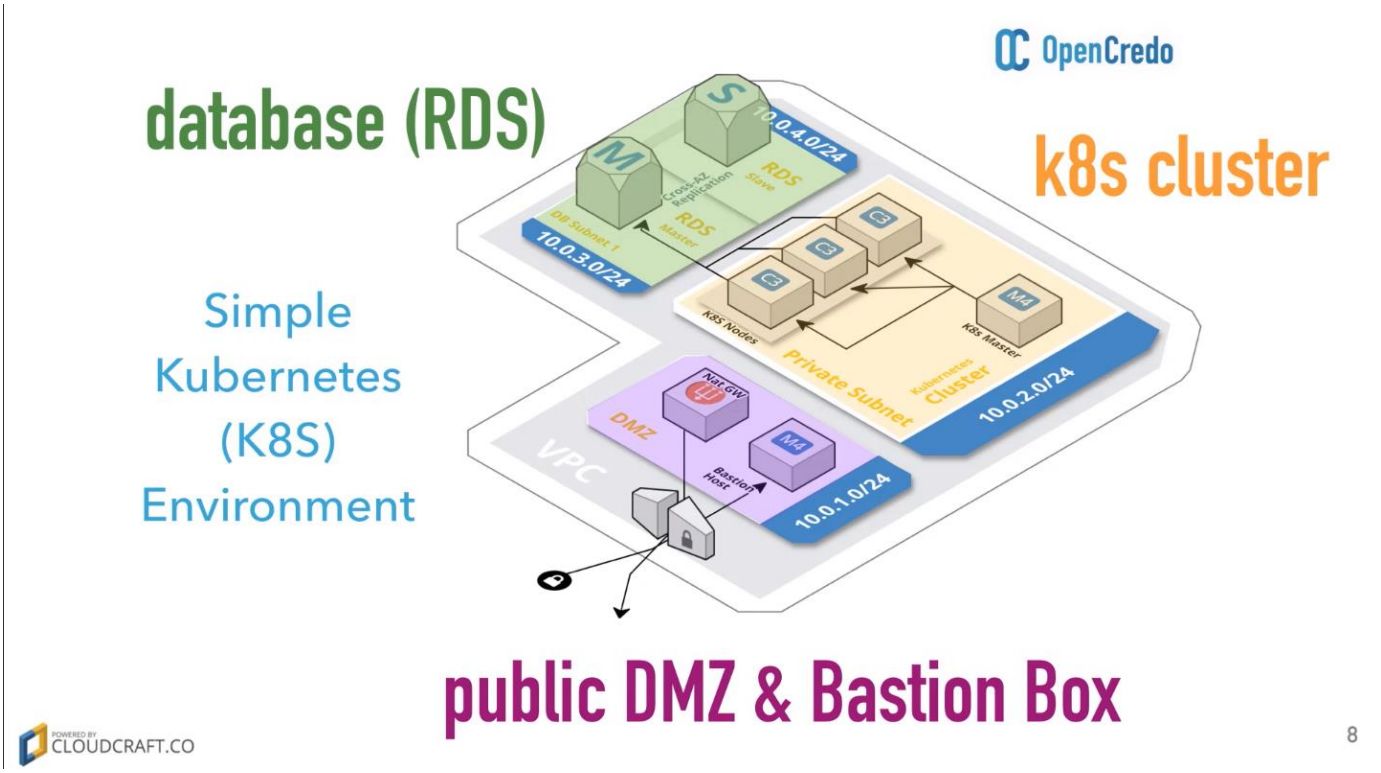
Example: E-Commerce System in AWS

(delivered as a Micro-services architecture)

The client is trying to deliver this microservices system in AWS and are planning to use terraform for creating the underlying infrastructure itself



They are going to be using K8s as the environment for actually deploying the microservices



The infra is relatively simple, we start off with creating a VPC that has a public subnet with a NAT gateway, Bastion host, we then have a private subnet that houses our K8s cluster with a single master node and three slave nodes to begin with. We are then going to be using Amazon RDS for our database needs in a DB subnet.

Pass #1 –
In the beginning . . .

Terri is a new DevOps engineer and she has just discovered programmable infrastructure in Terraform, she then creates a sample infra as below

https://github.com/mycompany/myproject

- terraform.tf
- terraform.tfvars
- terraform.tfstate

terraform.tf

```
## Test VPC
resource "aws_vpc" "test" {
    cidr_block           = "10.0.0.0/21"
    enable_dns_support   = true
    enable_dns_hostnames = true
}

## Staging Bastion
resource "aws_instance" “test_bastion" {
    ami           = "ami-7abd5555"
    instance_type = "t2.large"
    . . .
}
```

10

This has a single TF file ***terraform.tf*** that describes the resources that she wants to create, with some hard-coded values and some variables as above. She also creates a local ***terraform.tfstate*** file. She starts using this as her test infrastructure

We *must* to go to production this week …

Now she needs to set up a more formal environment for production

https://github.com/mycompany/myproject

- terraform.tf
- terraform.tfvars
- terraform.tfstate

terraform.tf

```
## Test VPC
resource "aws_vpc" "test" {
    cidr_block           = "10.0.0.0/21"
    enable_dns_support   = true
    enable_dns_hostnames = true
}

## Staging Bastion
resource "aws_instance" “test_bastion" {
    ami           = "ami-7abd5555"
    instance_type = "t2.large"
    . . .
}

## Prod VPC
resource "aws_vpc" "prod" {
    cidr_block           = "172.16.0.0/21"
    enable_dns_support   = true
    enable_dns_hostnames = true
}
```

12

She clones the test infra and duplicates that for the production setup as above.

https://github.com/mycompany/myproject

- terraform-prod.tf
- terraform-test.tf
- terraform.tfvars
- terraform.tfstate

terraform-test.tf

```
## Test VPC
resource "aws_vpc" "test" {
    cidr_block           = "10.0.0.0/21"
    enable_dns_support   = true
    enable_dns_hostnames = true
}
```

terraform-prod.tf

```
## Prod VPC
resource "aws_vpc" "prod" {
    cidr_block           = "172.16.0.0/21"
    enable_dns_support   = true
    enable_dns_hostnames = true
}

## Staging Bastion
resource "aws_instance" "prod_bastion" {
    ami           = "ami-7abd5555"
    instance_type = "t2.large"
    . . .
```

She then splits this into separate ***prod*** and ***test*** files for the terraform production setup ***terraform-prod.tf*** beside the original ***terraform-test.tf*** file as above. She runs the ***$ terraform apply*** command to stand up the test and prod infrastructure in AWS

Need an upgraded CIDR range in TEST …

We want to now increase the size of the K8s cluster and increase the CIDR range for the test VPC, she needs to make the change in the ***terraform-test.tf*** file for this

https://github.com/mycompany/myproject

- terraform-prod.tfbkp
- terraform-test.tf
- terraform.tfvars
- terraform.tfstate

terraform-test.tf

```
## Test VPC
resource "aws_vpc" "test" {
    cidr_block           = "10.0.0.0/21"
    enable_dns_support   = true
    enable_dns_hostnames = true
}
```

terraform-prod.tf

```
## Prod VPC
resource "aws_vpc" "prod" {
    cidr_block           = "172.16.0.0/21"
    enable_dns_support   = true
    enable_dns_hostnames = true
}

## Staging Bastion
resource "aws_instance" "prod_bastion" {
    ami           = "ami-7abd5555"
    instance_type = "t2.large"
    . . .
```

15

To make changes only to the test ***terraform-test.tf*** file, she renames the ***terraform-prod.tf*** file as above and runs the ***$ terraform plan*** and ***$ terraform apply*** commands

Help!
I seem to have deleted production

Terraform thought she deleted the prod resources and deleted them

"terralith"

17

This is because she has a monolithic configuration of keeping her state file for test and prod together

Terralith: Characteristics

- Single state file
- Single definition file
- Hard coded config
- Local state

Terralith – Pain points

- Can't manage environments separately
- Config not that intuitive (big ball of mud)
- Maintenance challenge: Duplicate Defs (not DRY)

Pass #2

She decides to get some help to evolve the state of the infrastructure

"multi terralith"

Multi Terralith: Characteristics

- Envs - Separate State Management
- Multiple Terraform Definition Files
- Better Use of Variables

Here we have separate environment state management in place, this helps reduce the risk from an operational perspective. It also uses multiple terraform definition files and start using variables a little bit better

https://github.com/mycompany/myproject

+ test
  - networks.tf
  - vms.tf
  - terraform.tfvars
  - terraform.tfstate

+ prod
  - networks.tf
  - vms.tf
  - terraform.tfvars
  - terraform.tfstate

networks.tf

```
resource "aws_vpc" "core" {
    cidr_block           = "${var.cidr}"
    enable_dns_support   = true
    enable_dns_hostnames = true
}
```

vms.tf

```
resource "aws_instance" "node" {
    count = "${var.node_count}"
    ami   = "ami-7abd5555"
    instance_type = "${var.vm_type}"
    . . .
}
```

23

We now have 2 different directories for test and prod with their separate files in them, we also have separate ***terraform.tfstate*** files in each folder for managing the different infrastructures.

https://github.com/mycompany/myproject

+ test
  - networks.tf
  - vms.tf
  - terraform.tfvars
  - terraform.tfstate

+ prod
  - networks.tf
  - vms.tf
  - terraform.tfvars
  - terraform.tfstate

networks.tf

```
resource "aws_vpc" "core" {
    cidr_block           = "${var.cidr}"
    enable_dns_support   = true
    enable_dns_hostnames = true
}
```

vms.tf

```
resource "aws_instance" "node" {
    count = "${var.node_count}"
    ami   = "ami-7abd5555"
    instance_type = "${var.vm_type}"
    . . .
}
```

We have also ***broken up the single terraform.tf file into multiple files like networks.tf and vms.tf*** (i.e. for networks and VMs) depending on what is in our environment.

https://github.com/mycompany/myproject

+ test
  - networks.tf
  - vms.tf
  - terraform.tfvars
  - *terraform.tfstate*

+ prod
  - networks.tf
  - vms.tf
  - terraform.tfvars
  - *terraform.tfstate*

networks.tf

```
resource "aws_vpc" "core" {
    cidr_block           = "${var.cidr}"
    enable_dns_support   = true
    enable_dns_hostnames = true
}
```

vms.tf

```
resource "aws_instance" "node" {
    count = "${var.node_count}"
    ami   = "ami-7abd5555"
    instance_type = "${var.vm_type}"
    . . .
}
```

25

We also have ***a separate variables file terraform.tfvars*** in each folder to define what aspects we want to make configurable within the environment instead of hard-coding everything.

Terralith - (recap)

- Can't manage environments separately
- Config not that intuitive (big ball of mud)
- Maintenance challenge: Duplicate Defs (not DRY)

Multi Terralith

✅ **Manage environment separately**
(separate state files per env)

🔶 **More intuitive configuration**
(multiple files)

- Maintenance challenge: Duplicate Defs
(not DRY)

Pass #3

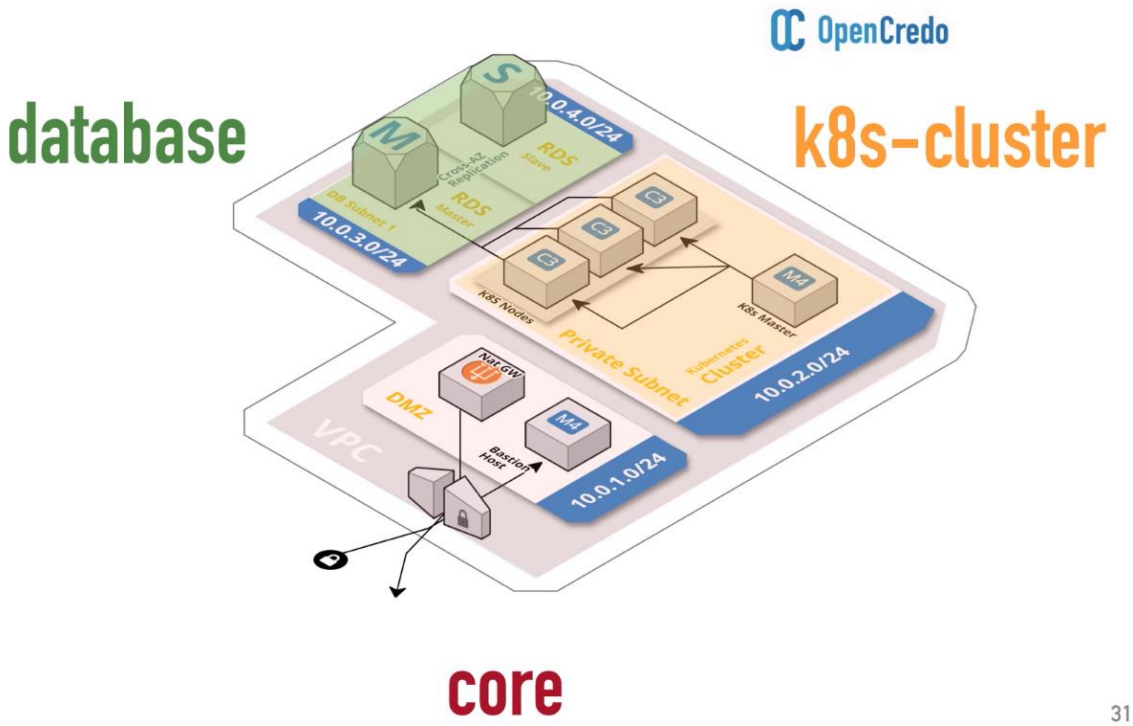
"terramod"

We are now evolving into a version of terraform that looks to make use of Modules, this will allow us to create re-useable components that we can start to compose our infrastructure from. Terraform ahs built-in support for modules and we are going to be using modules as the base blocks to evolve our infrastructure out of.

Terramod: Characteristics

- Reusable modules
- Envs compose themselves from modules
- Restructuring of repo

We are going to change our environment configuration to start composing themselves out of the modular definitions that we are going to be creating. We are also going to change our repo structure as we go along as well.

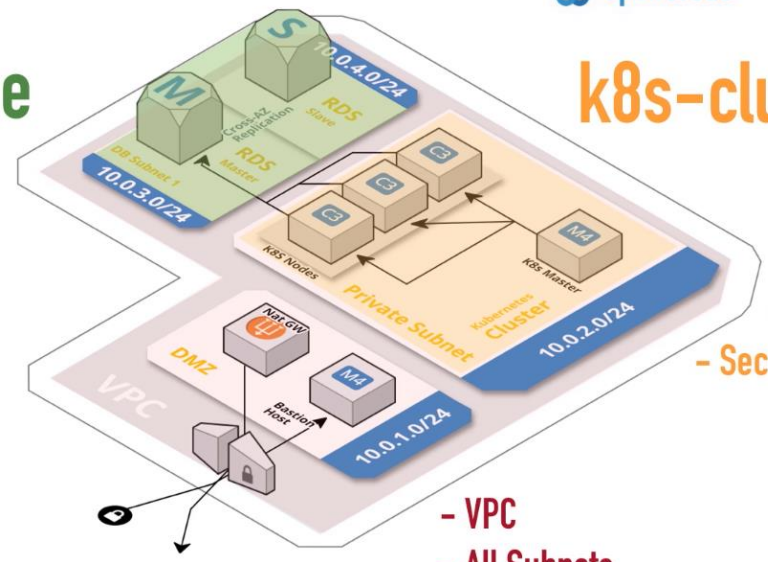


We are going to logically break up our modules into the 3 pieces above.

OpenCredo

database

- Amazon RDS
- DB Subnet Group



k8s-cluster

- Instances
- Security Groups

core

- VPC
- All Subnets
- Core Routing & Gateways
- Bastion Host (OpenVPN server)

POWERED BY CLOUDCRAFT.CO

32

https://github.com/mycompany/myproject

```
+ envs/[test|prod]
    - config.tf
    - terraform.tf
    - terraform.tfvars
    - terraform.tfstate

+ modules
  + core
    - input.tf
    - core.tf
    - output.tf
  + k8s-cluster
    - input.tf
    - dns.tf
    - vms.tf
    - output.tf
```

separate env management & module defs

33

We now have ***an environments directory*** called ***envs*** with 2 ***sub-folders*** for ***test*** and ***prod***.

https://github.com/mycompany/myproject

```
+ envs/[test|prod]
    - config.tf
    - terraform.tf
    - terraform.tfvars
    - terraform.tfstate

+ modules
  + core
    - input.tf
    - core.tf
    - output.tf
  + k8s-cluster
    - input.tf
    - dns.tf
    - vms.tf
    - output.tf
```

define logical components as re-usable modules

34

We then start to define the logical components divided up by the 3 areas ***core***, ***k8s-cluster***, and ***database***.

https://github.com/mycompany/myproject

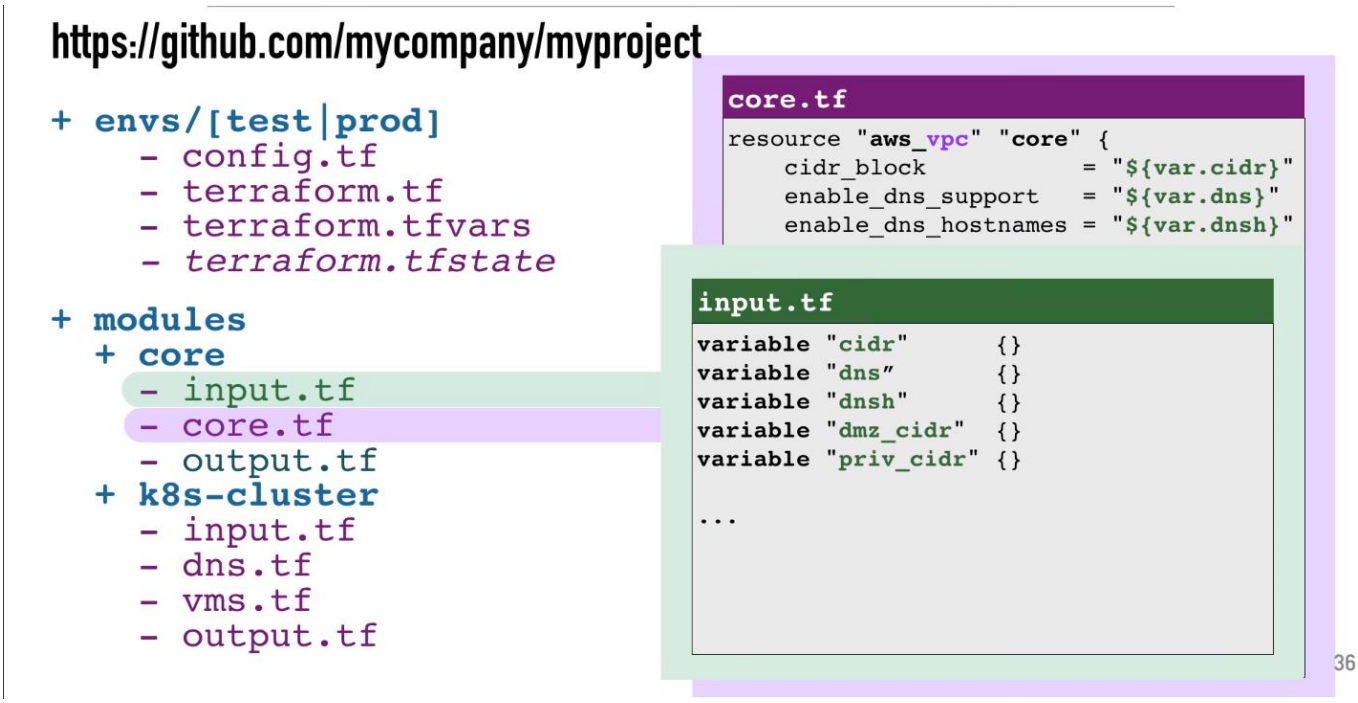
```
+ envs/[test|prod]
    - config.tf
    - terraform.tf
    - terraform.tfvars
    - terraform.tfstate

+ modules
  + core
    - input.tf
    - core.tf
    - output.tf
  + k8s-cluster
    - input.tf
    - dns.tf
    - vms.tf
    - output.tf
```

core.tf

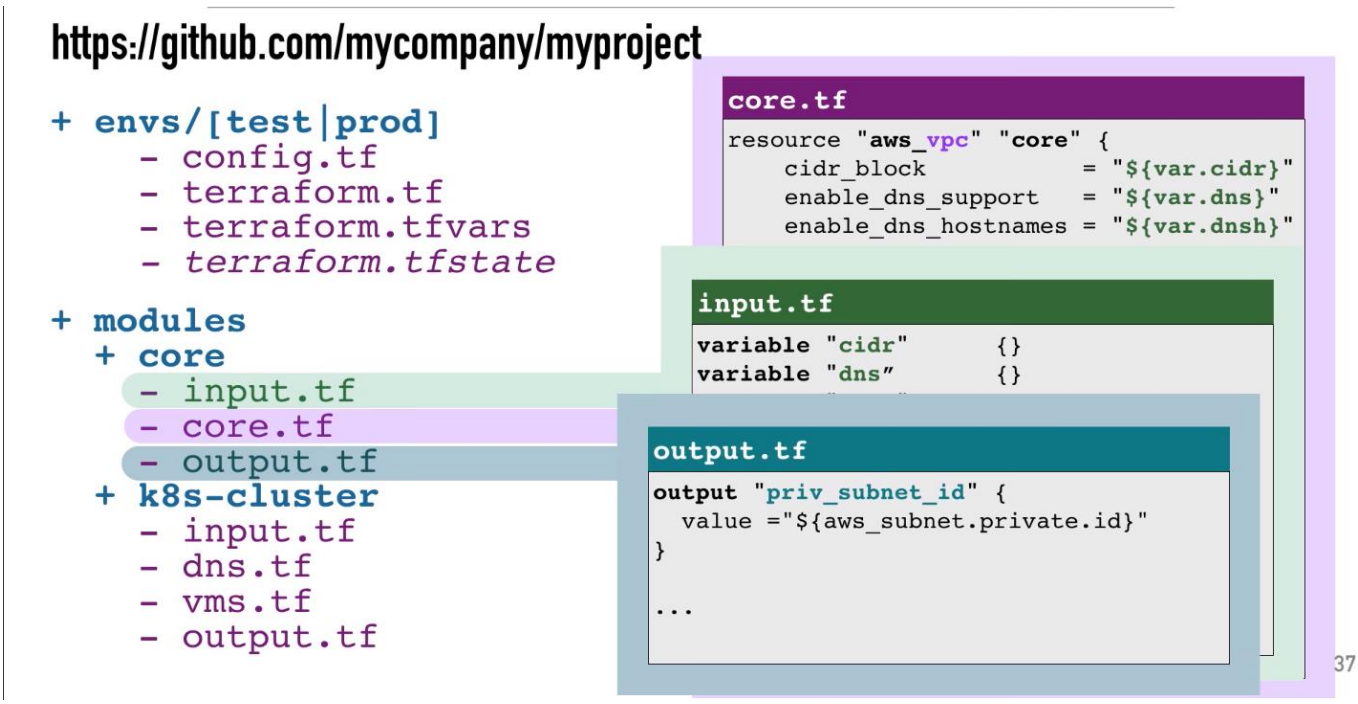
```
resource "aws_vpc" "core" {
    cidr_block           = "${var.cidr}"
    enable_dns_support   = "${var.dns}"
    enable_dns_hostnames = "${var.dnsh}"
}

resource "aws_subnet" "dmz" {
    vpc_id     = "${aws_vpc.core.id}"
    cidr_block = "${var.dmz_cidr}"
    map_public_ip_on_launch = 1
    ...
}

resource "aws_subnet" "private" {
    vpc_id     = "${aws_vpc.core.id}"
    cidr_block = "${var.priv_cidr}"
    ...
}
```

35

For each module, we try to define all the resources that make up the creation of that particular piece.



We also try to have a very clear contract that defines what are the inputs and outputs that make up that particular module. The ***input.tf*** file should contain the variables that we want to configure that module with and the ***output.tf*** file should contain the expected output values.



The output.tf file contains the expected outputs that we want to have to use in other modules by referring to their output names during module composition.

https://github.com/mycompany/myproject

```
+ envs/[test|prod]
    - config.tf
    - terraform.tf
    - terraform.tfvars
    - terraform.tfstate

+ modules
  + core
    - input.tf
    - core.tf
    - output.tf
  + k8s-cluster
    - input.tf
    - dns.tf
    - vms.tf
    - output.tf
```

defines the contract of the module

38

https://github.com/mycompany/myproject

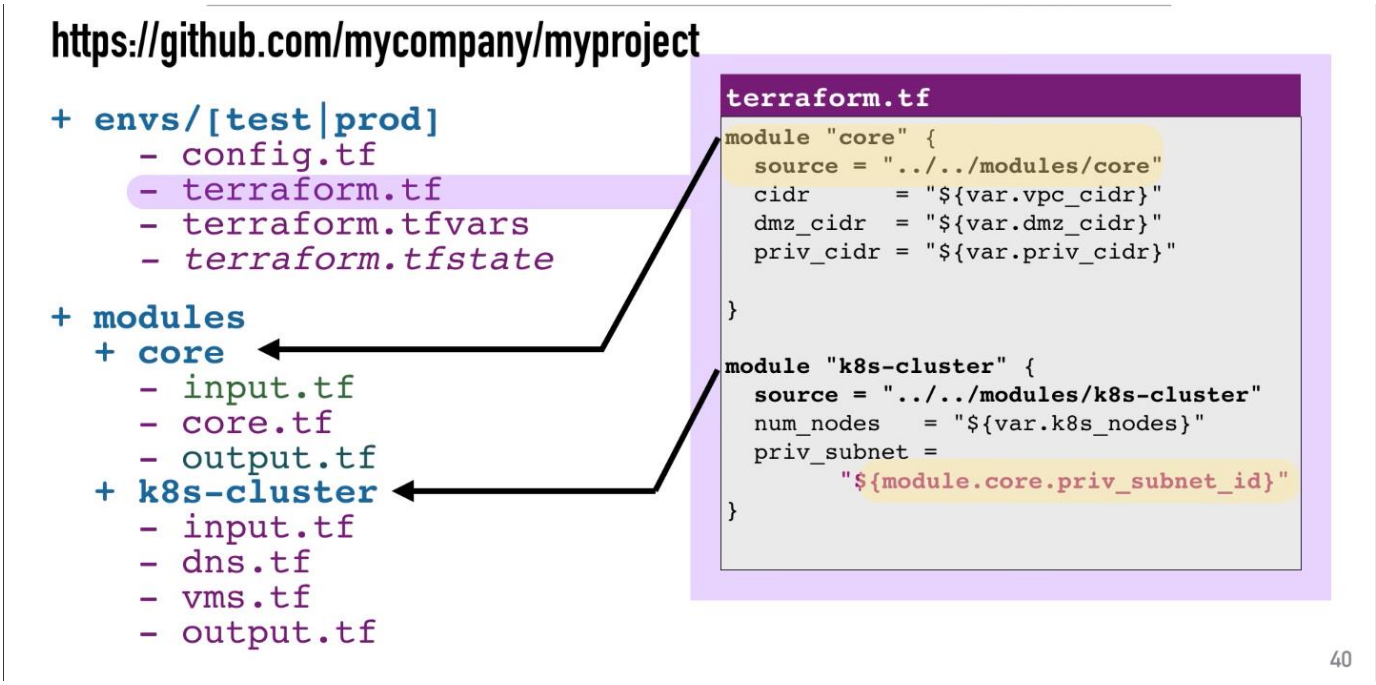
```
+ envs/[test|prod]
    - config.tf
    - terraform.tf
    - terraform.tfvars
    - terraform.tfstate

+ modules
  + core
    - input.tf
    - core.tf
    - output.tf
  + k8s-cluster
    - input.tf
    - dns.tf
    - vms.tf
    - output.tf
```

terraform.tf

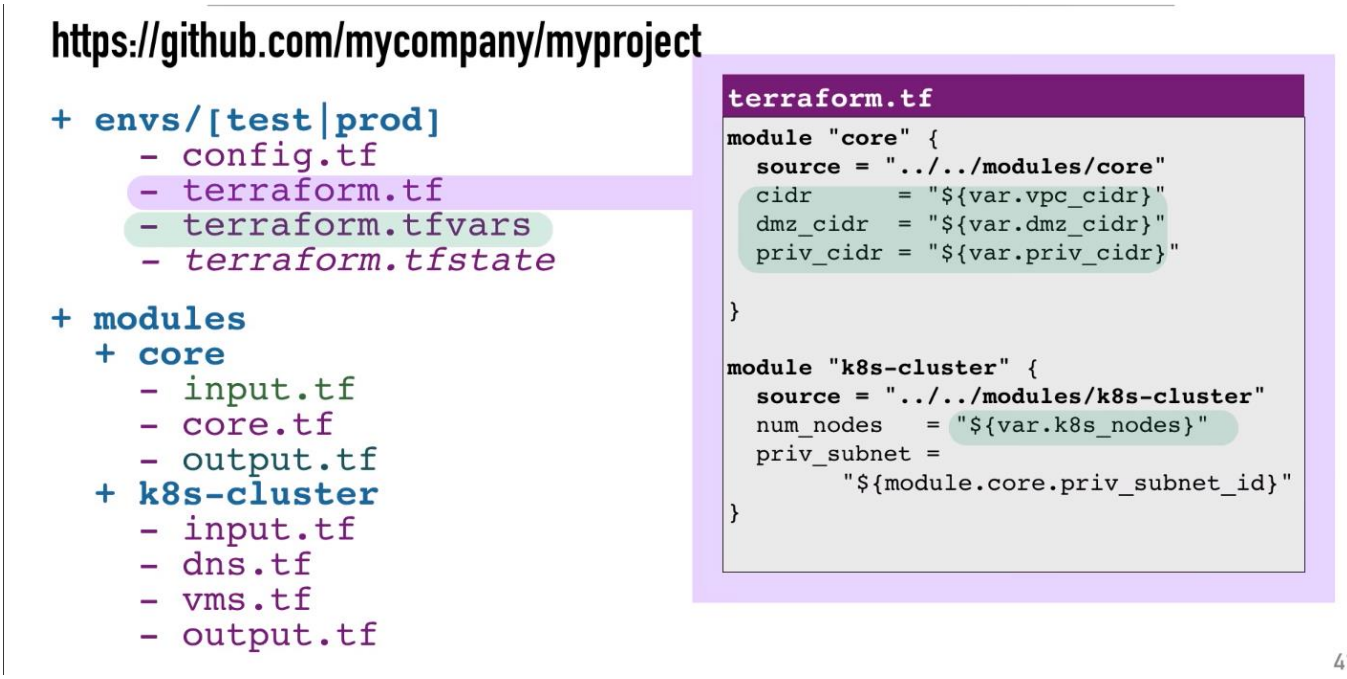
```
module "core" {
  source = "../../modules/core"
  cidr       = "${var.vpc_cidr}"
  dmz_cidr   = "${var.dmz_cidr}"
  priv_cidr  = "${var.priv_cidr}"

}

module "k8s-cluster" {
  source = "../../modules/k8s-cluster"
  num_nodes    = "${var.k8s_nodes}"
  priv_subnet =
        "${module.core.priv_subnet_id}"
}
```

39

For each core environment, the terraform file that we have in them now become a sort of gluing module.



We also have to starting weaving the outputs from one module into other modules that might also need them as inputs. We are using the private subnet value created in the core module as an output to be used when creating our private subnet in the k8s cluster module as input so that we can make sure it gets created in the correct subnet.



Because all of the modules are configurable, there is a very clear contract which means that for different environments, we can start configuring things differently. E.g. we might need only 3 nodes for the k8s cluster in our test environment, but we want to use 5 nodes in production. We can use different variables that help configure things differently for each environment.

Multi Terralith

✅ **Manage environment separately**
(separate state files per env)

🔶 **More intuitive configuration**
(multiple files)

- Maintenance challenge: Duplicate Defs
(not DRY)

Terramod

✅ **Manage environment separately**
(separate state files per env)

✅ **Intuitive configuration**
(reusable modules)

🔶 **Reduced Duplicate Definitions**
(DRYer)

We are now composing our environments out of same set of modules but just passing in different values.

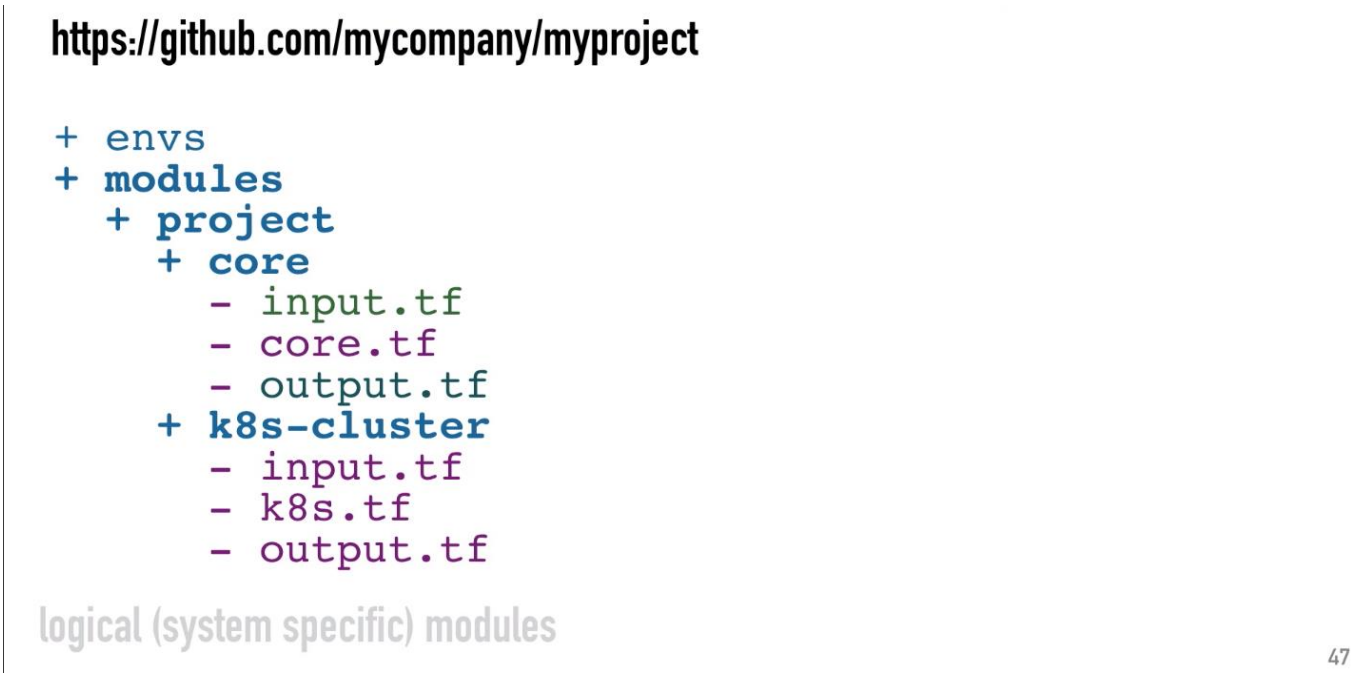
Pass #4

terramod^n

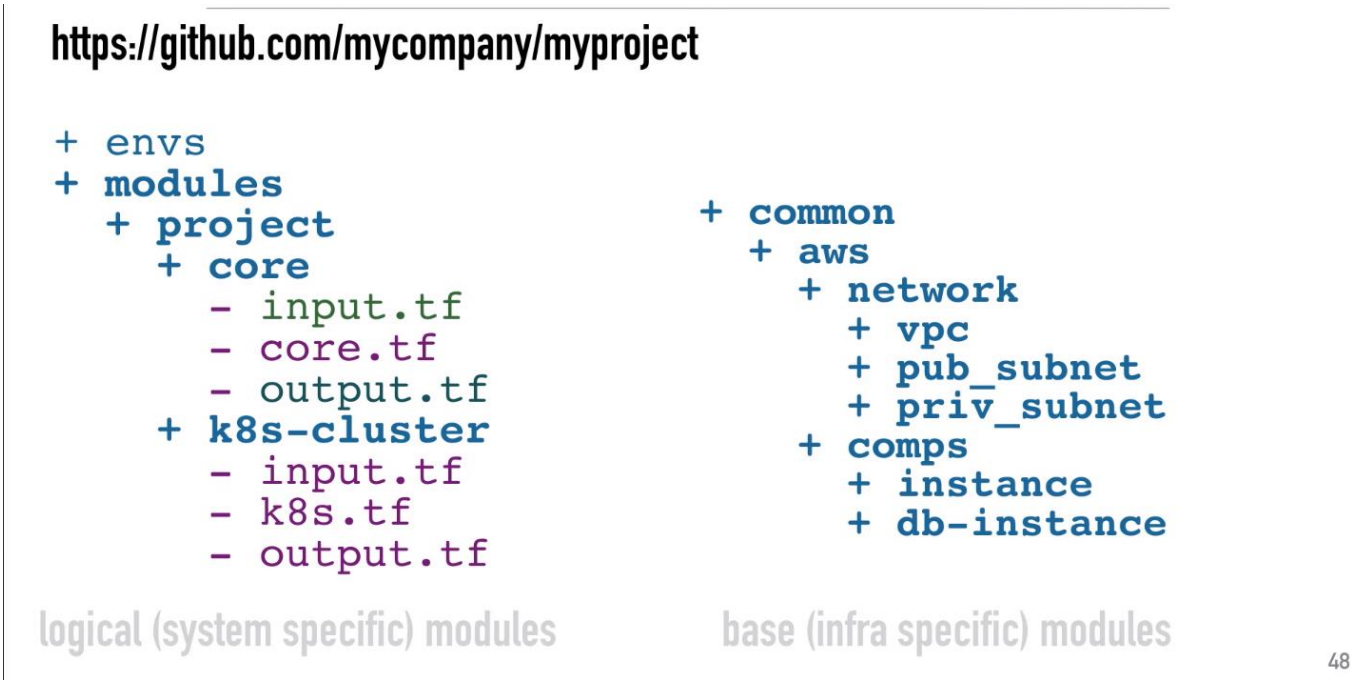
This builds on and takes the use of modules to a new level where we end up having nested modules or modules within modules.

Terramod^n : Characteristics

- Nested modules
  - base modules
    (low level infrastructure specific)
  - logical modules
    (system specific)
- Sometimes dedicated module repo



This is where we are at the moment



Now we simply just add these base modules as well. We can create low level modules that specifies exactly how we create a VPC, or how we exactly create public and private subnets in AWS.

https://github.com/mycompany/myproject

```
+ envs
+ modules
  + project
    + core
      - input.tf
      - core.tf
      - output.tf
    + k8s-cluster
      - input.tf
      - k8s.tf
      - output.tf
```

modules/project/core/core.tf

```
resource "aws_vpc" "core" {
    cidr_block           = "${var.cidr}"
    enable_dns_support   = "${var.dns}"
    enable_dns_hostnames = "${var.dnsh}"
}

resource "aws_subnet" "dmz" {
    vpc_id     = "${aws_vpc.core.id}"
    cidr_block = "${var.dmz_cidr}"
    map_public_ip_on_launch = 1
    ...
}
```


This is what our core module looks like previously, where we have all the direct resources like the VPC and subnets being defined in there as above.

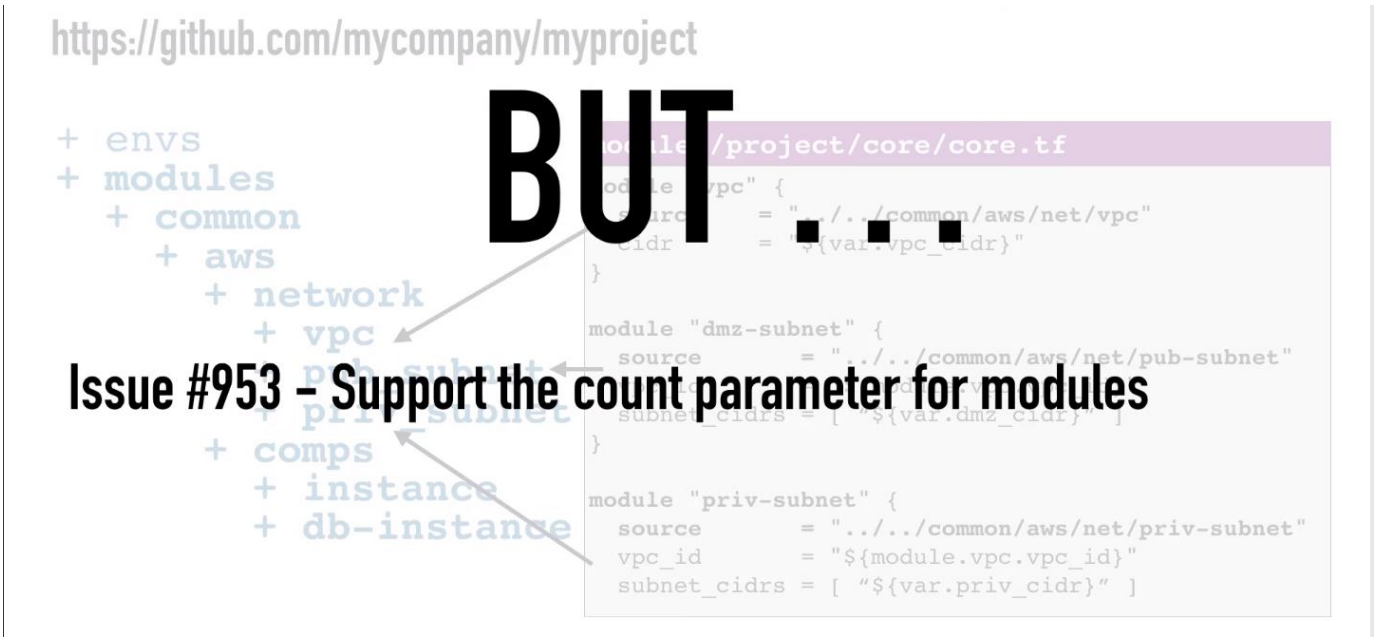
https://github.com/mycompany/myproject

```
+ envs
+ modules
  + common
    + aws
      + network
        + vpc
        + pub_subnet
        + priv_subnet
      + comps
        + instance
        + db-instance
```

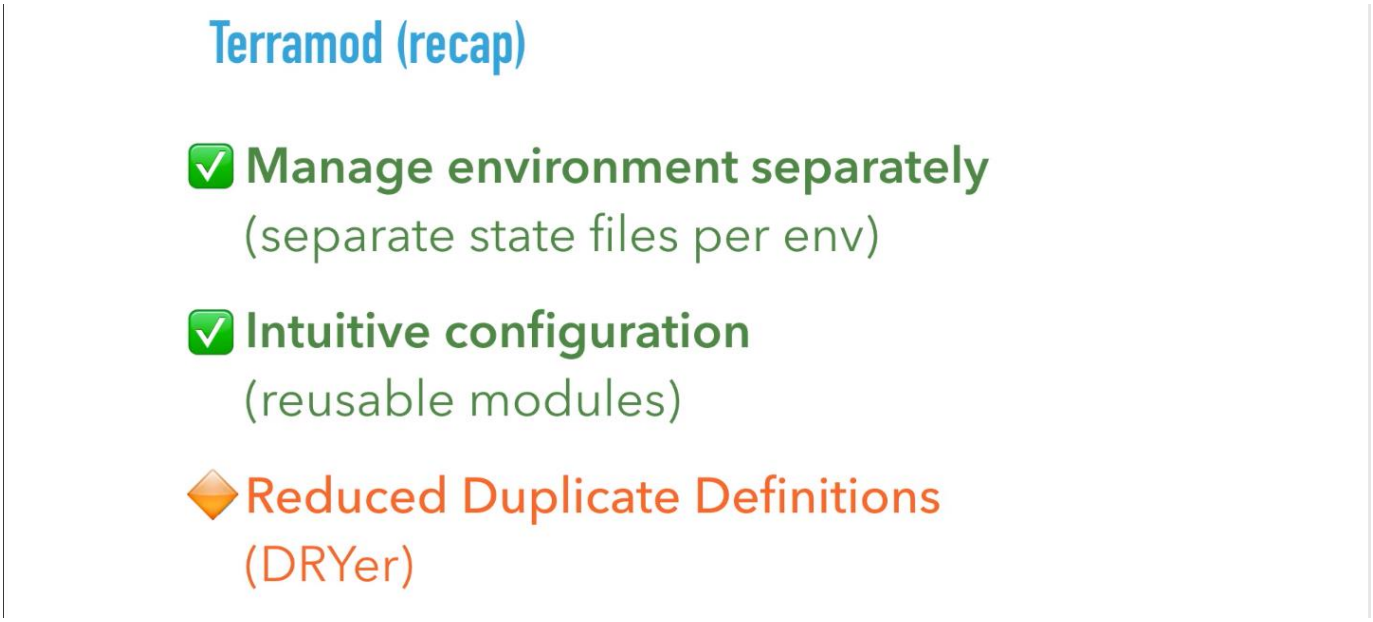
modules/project/core/core.tf

```
module "vpc" {
  source     = "../../common/aws/net/vpc"
  cidr       = "${var.vpc_cidr}"
}

module "dmz-subnet" {
  source       = "../../common/aws/net/pub-subnet"
  vpc_id       = "${module.vpc.vpc_id}"
  subnet_cidrs = [ "${var.dmz_cidr}" ]
}

module "priv-subnet" {
  source       = "../../common/aws/net/priv-subnet"
  vpc_id       = "${module.vpc.vpc_id}"
  subnet_cidrs = [ "${var.priv_cidr}" ]
```

This changes the direct resources themselves to now being composed of modules as above, we now have our core modules being composed of our base modules. We can also compose system modules from other system modules.



Terraform prevents you from supporting a count parameter for modules, we can't say we want N amount of modules. We have to duplicate the modules in the environment configuration files ourselves



Terramod^n

✅ **Manage environment separately**
(separate state files per env)

✅ **Intuitive configuration**
(reusable modules)

🔶 **Reduced Duplicate Definitions further**
(as DRY as possible given restrictions)

Time goes on …

Maintenance required …
- Make bastion box smaller -

```
+ envs/prod
    - config.tf
    - terraform.tf
    - terraform.tfvars
    - terraform.tfstate
```

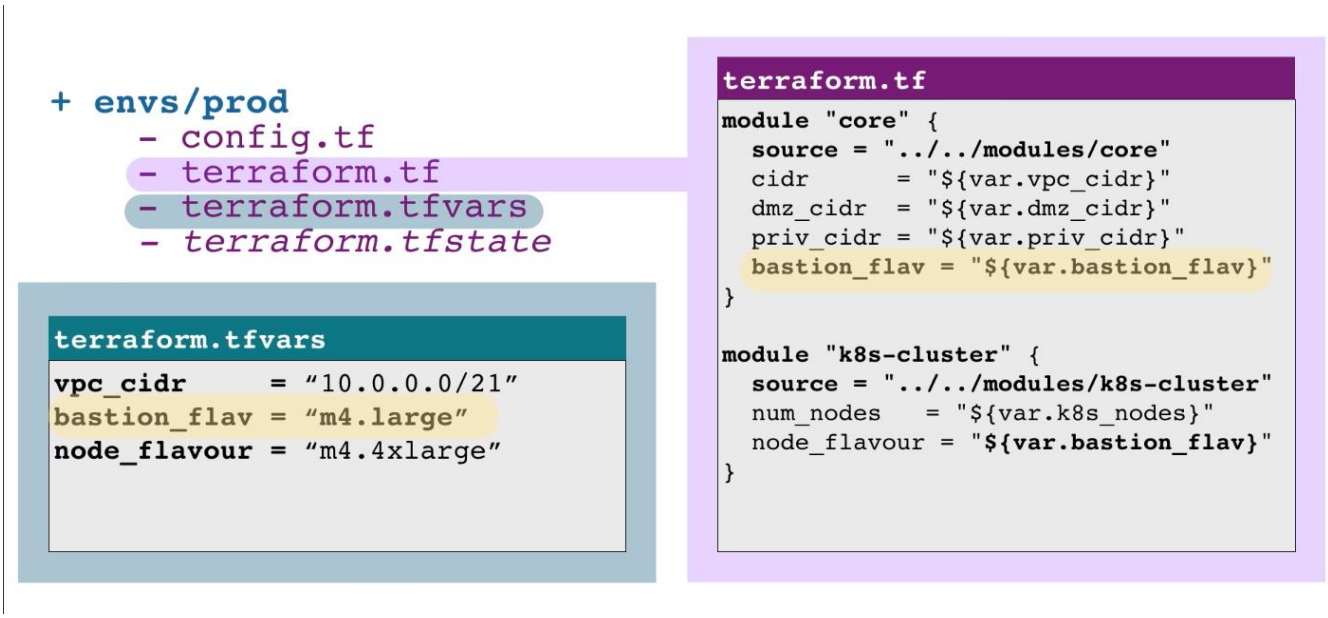
terraform.tfvars

```
vpc_cidr      = "10.0.0.0/21"
bastion_flav  = "r4.large"
node_flavour  = "m4.4xlarge"
```

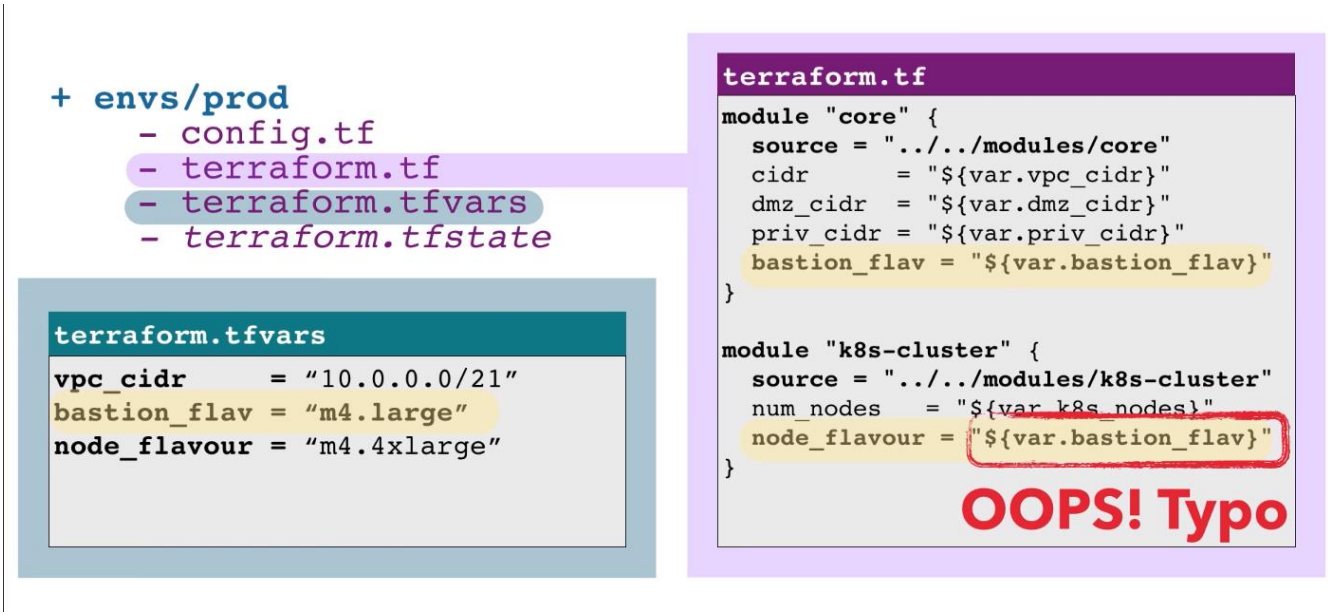
terraform.tf

```
module "core" {
  source = "../../modules/core"
  cidr       = "${var.vpc_cidr}"
  dmz_cidr   = "${var.dmz_cidr}"
  priv_cidr  = "${var.priv_cidr}"
  bastion_flav = "${var.bastion_flav}"
}

module "k8s-cluster" {
  source = "../../modules/k8s-cluster"
  num_nodes    = "${var.k8s_nodes}"
  node_flavour = "${var.bastion_flav}"
}
```

We simply locate the variable that specifies the instance that we are using for the bastion host and change it as below



If we just do a ***$ terraform apply*** command, we get some issues

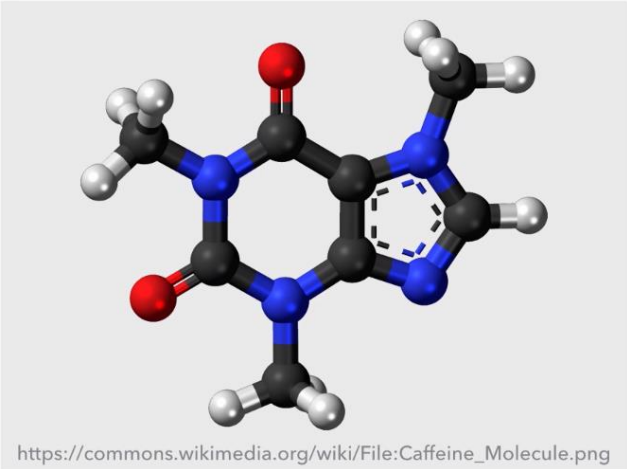


This is because the same variable used to configure the bastion host was being used in the configuration of the k8s cluster also. You need to do a ***$ terraform plan*** before doing a ***$ terraform apply*** command

Next set of pain!

- Can't manage logical parts of our infrastructure independently

Pass #5



https://commons.wikimedia.org/wiki/File:Caffeine_Molecule.png

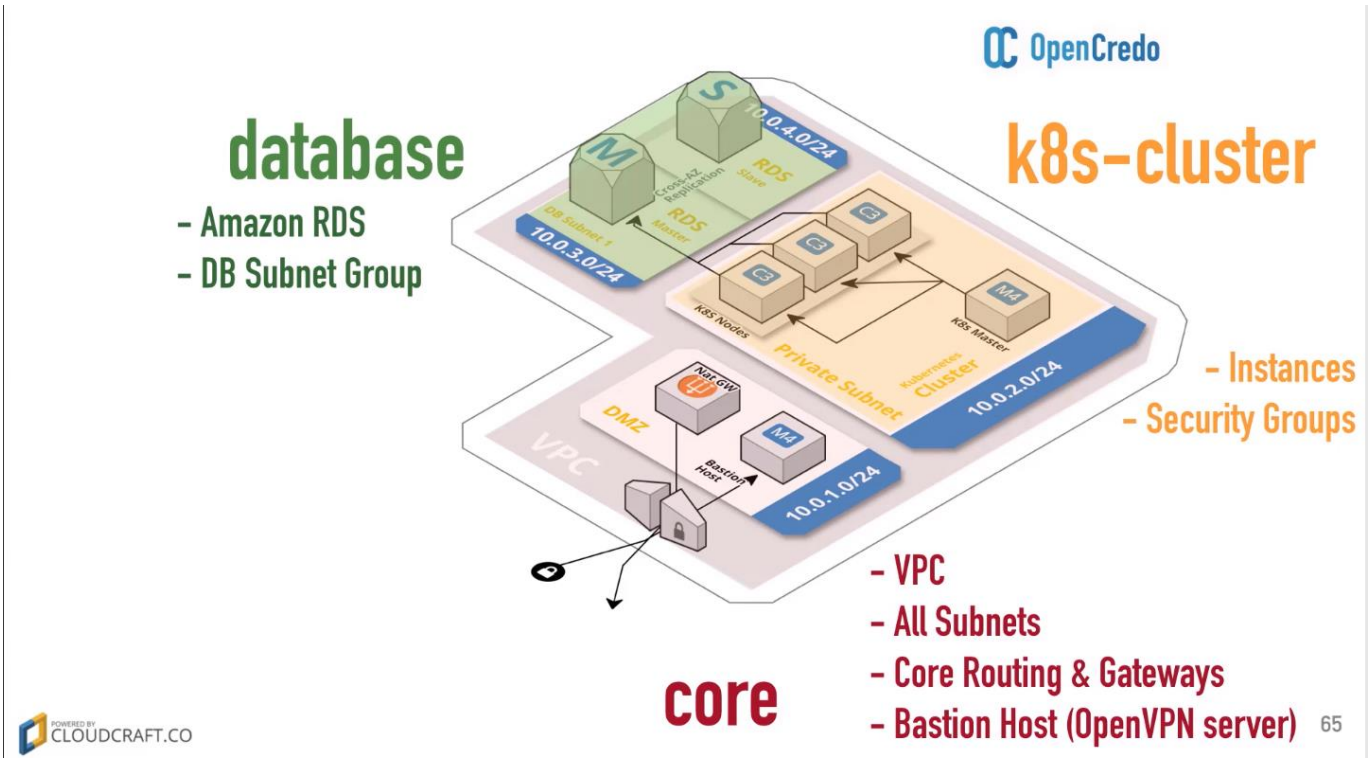
"terraservices"

This looks at taking the logical components we had before and treating them as isolated unit and managing them independently. This would help isolate the risk and management of things affecting each other

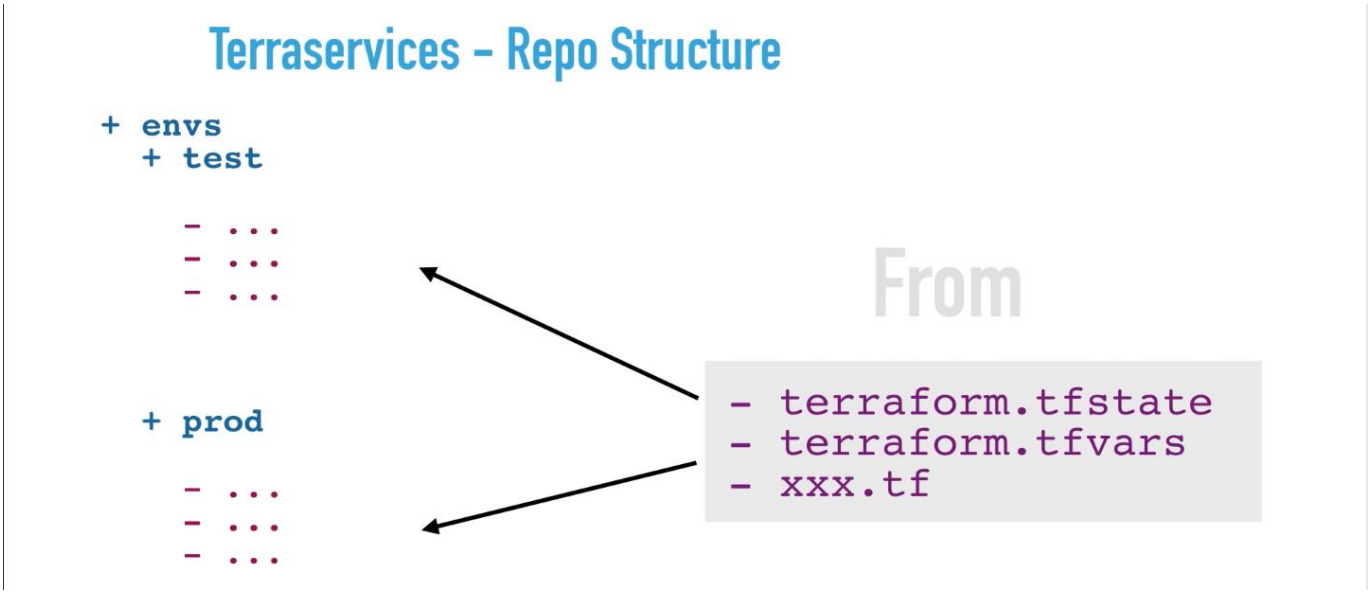
Terraservices - Characteristics

- Independent management of logical comps
  - Isolates & Reduces Risk
  - Aids with Multi Team Setups
- Distributed (Remote State)
- Requires additional orchestration effort

Now we move to having one state file per component rather than per environment.



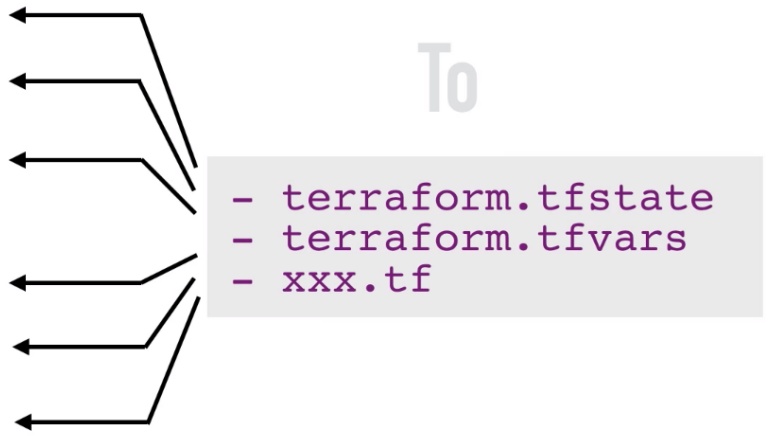
Recall that even though we now have 3 main modules, we are still being rules by a single environment state file



We now have to start having one state file for each of the module components, instead of the 2 state files above

Terraservices - Repo Structure

```
+ envs
  + test
    + core
      - ...
    + database
      - ...
    + k8s-cluster
      - ...

  + prod
    + core
      - ...
    + database
      - ...
    + k8s-cluster
      - ...
```



67

We end up having 6 state environment files in this new terraservices approach

Terramod - Connecting (recap)

```
+ envs
  + test

    - ...
    - ...
    - ...
```

From

envs/test/terraform.tf

```
module "core" {
  source = "../../modules/core"
  cidr       = "${var.vpc_cidr}"
  dmz_cidr   = "${var.dmz_cidr}"
  priv_cidr = "${var.priv_cidr}"

}

module "k8s-cluster" {
  source = "../../modules/k8s-cluster"
  num_nodes    = "${var.k8s_nodes}"
  priv_subnet =
        "${module.core.priv_subnet_id}"
}
```

Connecting things together now needs to change, previously we have the above way of using outputs from other modules as inputs in other modules as above for the Terramod configuration

Terraservices - Connecting

```
+ envs
  + test
    + core
      - ...
    + database
      - ...
    + k8s-cluster
      - ...
```

To

envs/test/core/terraform.tf

```
# Optional but explicit! (Needs 0.9+)
terraform {
   backend "local" {
      path = "terraform.tfstate"
   }
}

module "core" {
  source = "../../modules/core"
  cidr       = "${var.vpc_cidr}"
```

envs/test/core/outputs.tf

```
output "priv_subnet_id" {
  value ="${module.core.priv_subnet_id}"
}
```

69

This is what we now have

Terraservices - Connecting

```
+ envs
  + test
    + core
      - ...
    + database
      - ...
    + k8s-cluster
      - ...
```

To

envs/test/core/terraform.tf

envs/test/k8s-cluster/terraform.tf

```
data "terraform_remote_state" "core" {
  backend = "local"
  config {
    path = "../core/terraform.tfstate"
  }
}

module "k8s-cluster" {
  source = "../../modules/k8s-cluster"
  num_nodes    = "${var.k8s_nodes}"
  priv_subnet = "${data.terraform_remote_
                  state.core.priv_subnet_id}"
  }
}
```

70

This is how we configure the components that now want to consume other components, we need to import the component we want to connect to by using the ***data.terraform_remote_state*** as above

Terraservices - Characteristics

- Independent management of logical comps
  - Isolates & Reduces Risk
  - Aids with Multi Team Setups
- Distributed (Remote State)
- Requires additional orchestration effort

Terraservices - Distributed (Remote State)

```
+ envs
  + test
    + core
      - ...
    + database
      - ...
    + k8s-cluster
      - ...
```

From

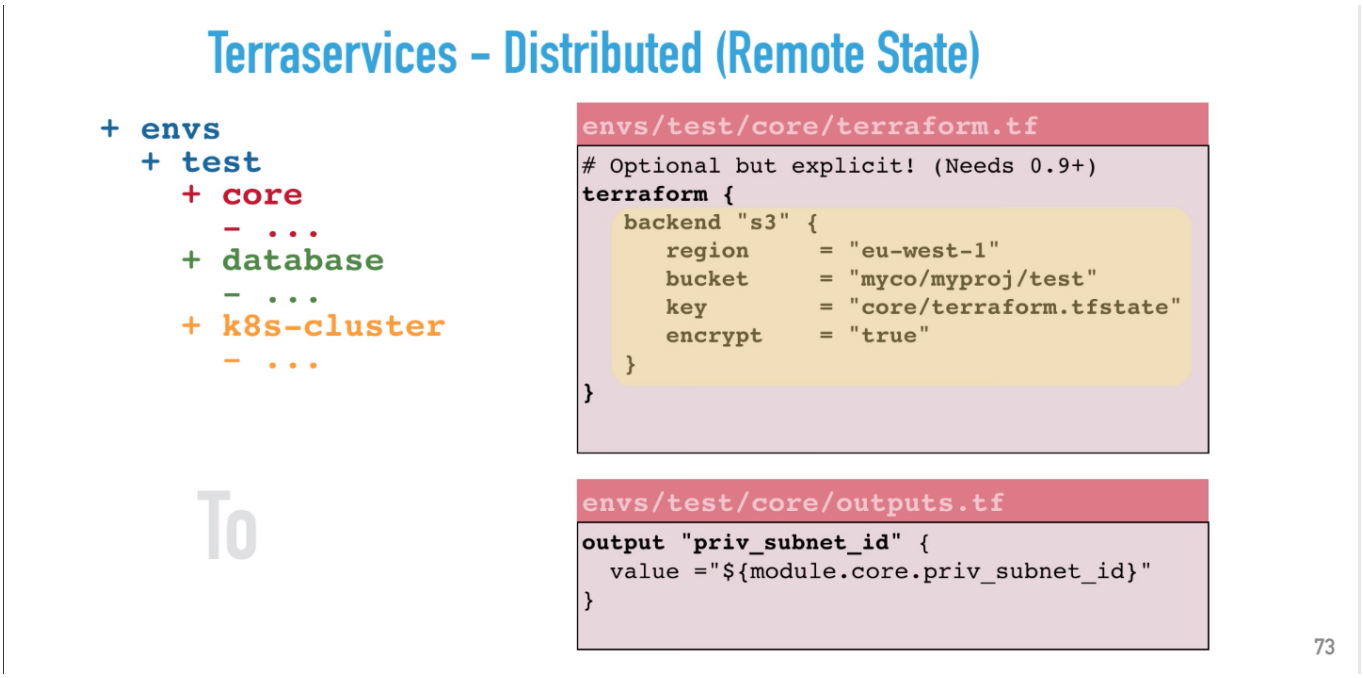
envs/test/core/terraform.tf

```
# Optional but explicit! (Needs 0.9+)
terraform {
   backend "local" {
      path = "terraform.tfstate"
   }
}

module "core" {
  source = "../../modules/core"
  cidr       = "${var.vpc_cidr}"
```

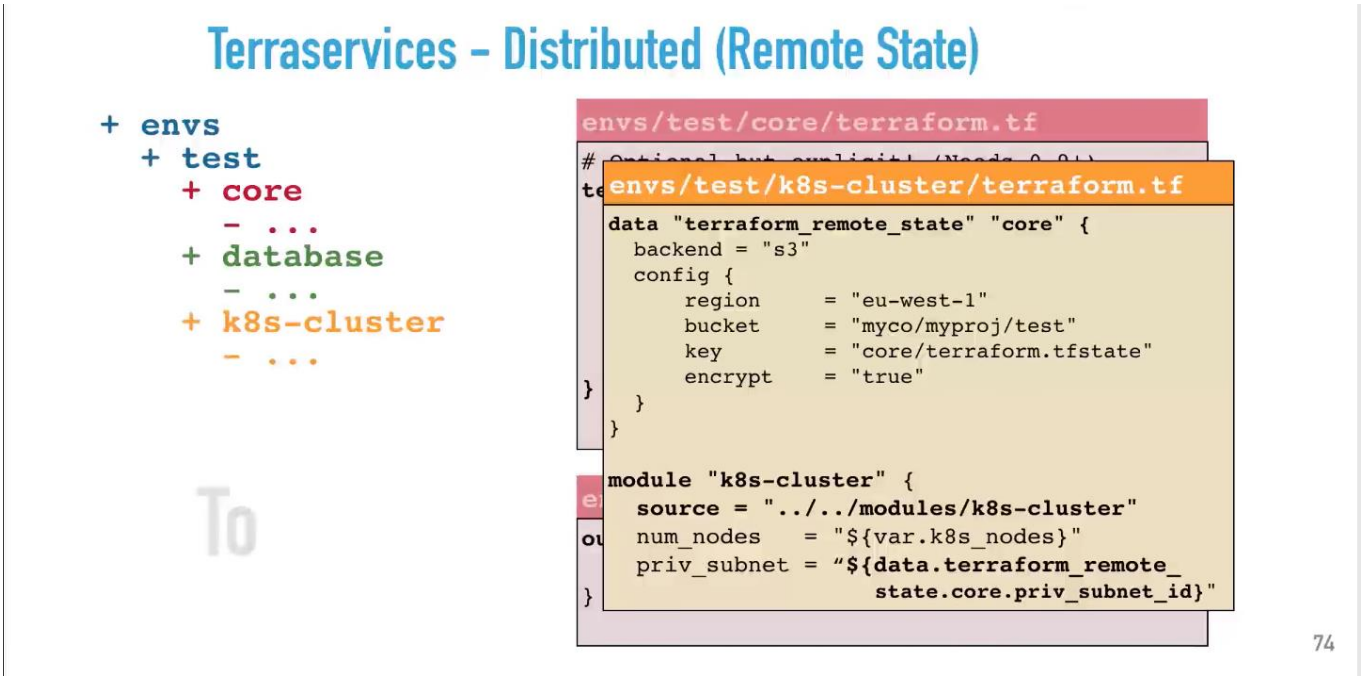
envs/test/core/outputs.tf

```
output "priv_subnet_id" {
  value ="${module.core.priv_subnet_id}"
}
```


Previously we have the local reference to the terraform state file,



Now we can simply change the backend to something else that we want to use like Amazon S3 as the new backend, replacing the local backend.



This helps from a team perspective now. We also can encrypt S3 to protect our secrets at rest

Terraservices – Repo Isolation (Optional)

https://github.com/myco/myproj

From

```
+ envs
  + test|prod
    + core
      - ...
    + database
      - ...
    + k8s-cluster
      - ...

+ modules
  + common
    + aws
      + network
        + vpc
```

75

We might have different teams responsible for creating different modules

Terraservices – Repo Isolation (Optional)

https://github.com/myco/myproj

To

```
+ envs
  + test|prod
    + core                    https://github.com/myco/myproj-core
      - ...
    + database                https://github.com/myco/myproj-db
      - ...
    + k8s-cluster             https://github.com/myco/myproj-k8s
      - ...

+ modules                     https://github.com/myco/tf-modcomm
  + common
    + aws
      + network
        + vpc
```

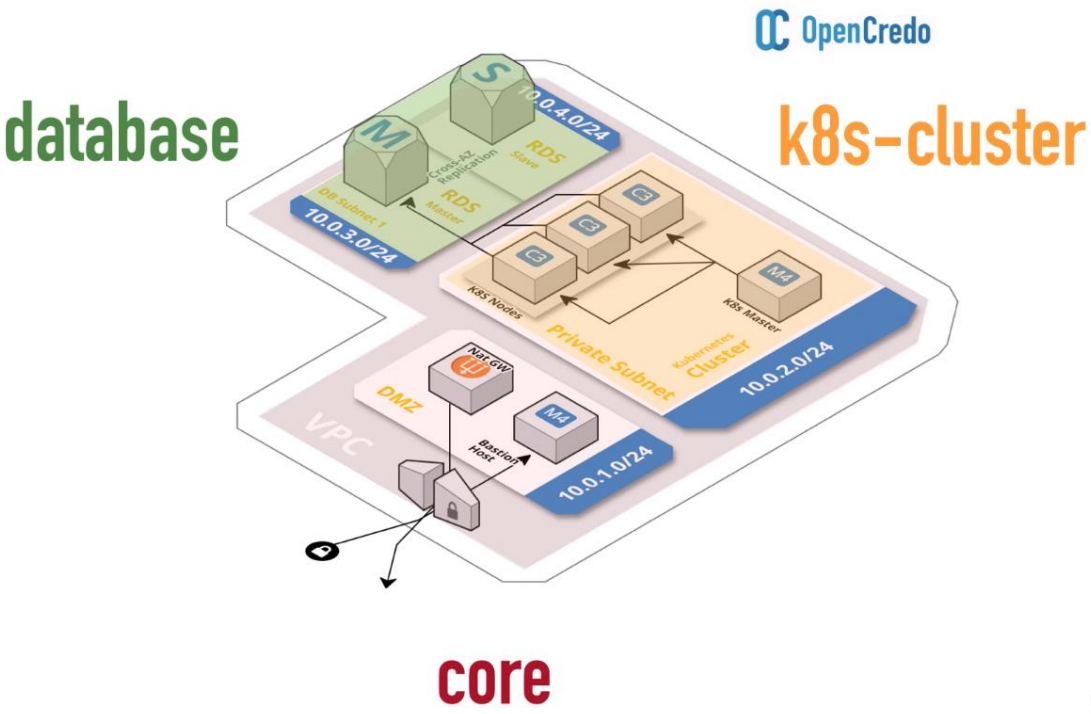
76

We then can let them manage their repos separately as above

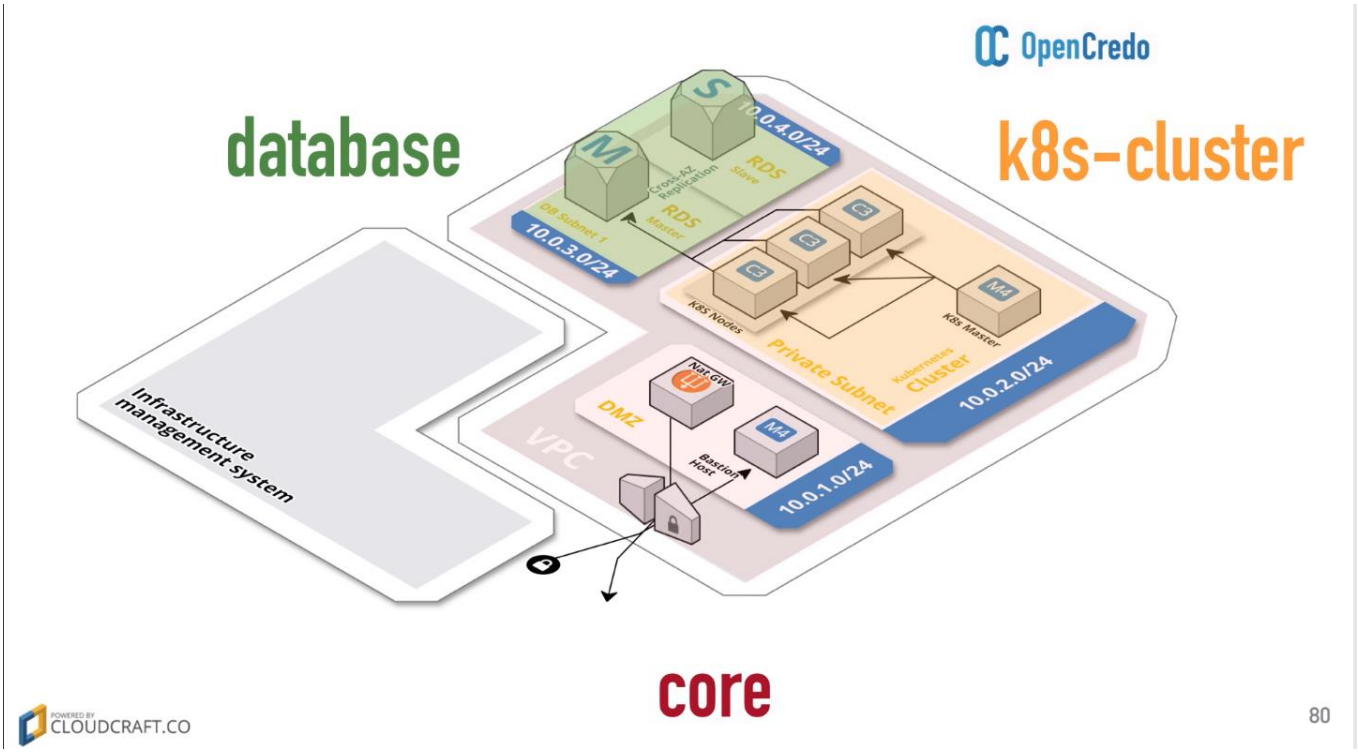
Terraservices - Characteristics

- Independent management of logical comps
  - Isolates & Reduces Risk
  - Aids with Multi Team Setups
- Distributed (Remote State)
- Requires additional orchestration effort

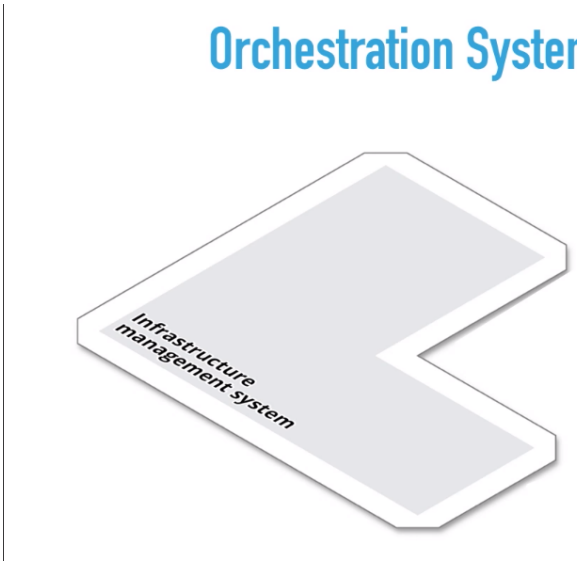
Orchestrating your Terraform



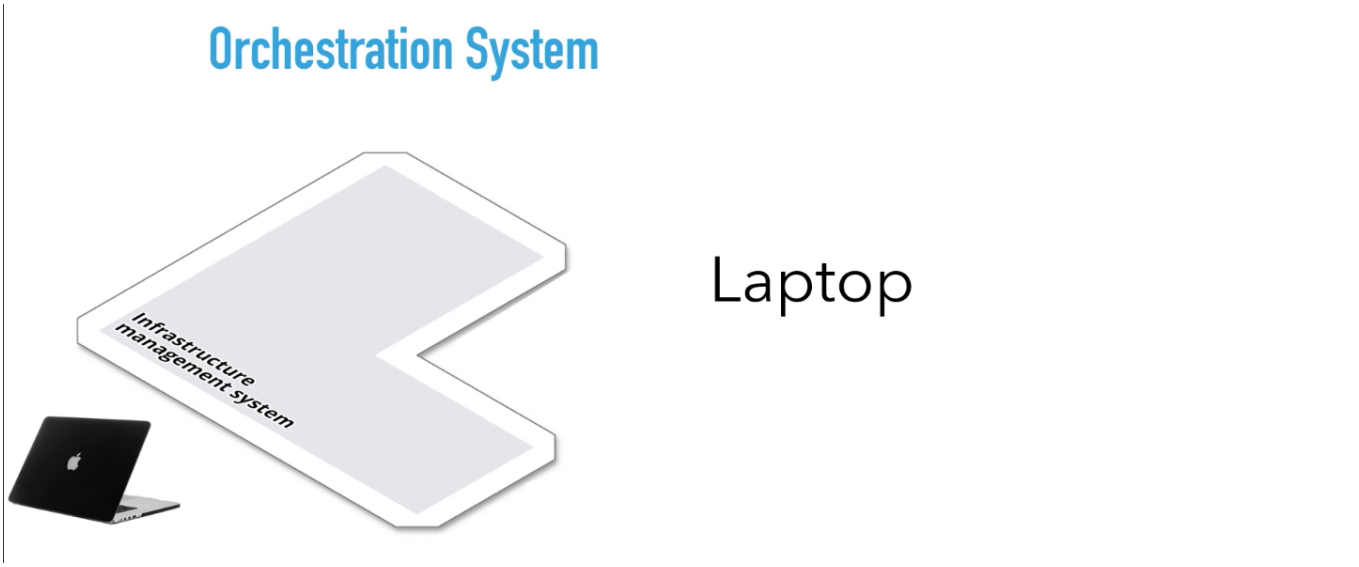
This was our target infrastructure



We also ended up having some kind of system or process or tooling for managing and orchestrating terraform.

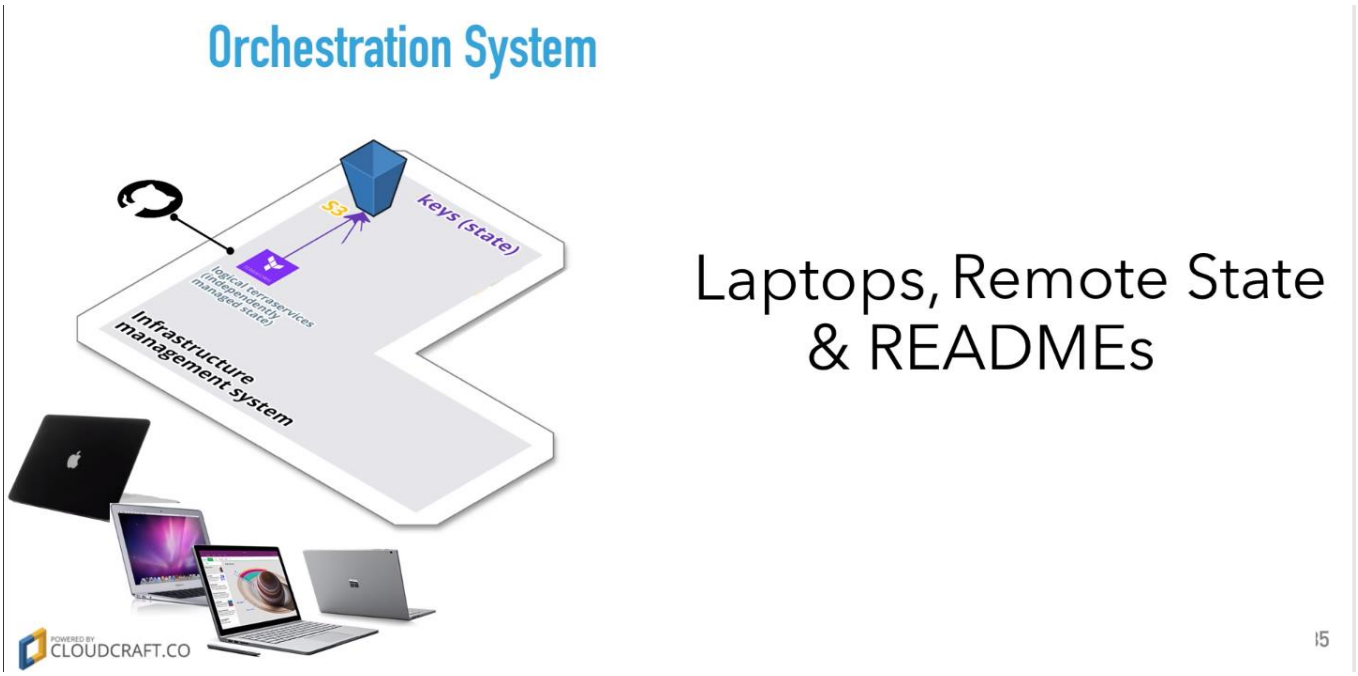


How do we evolve the processes that help us in managing terraform as a team?

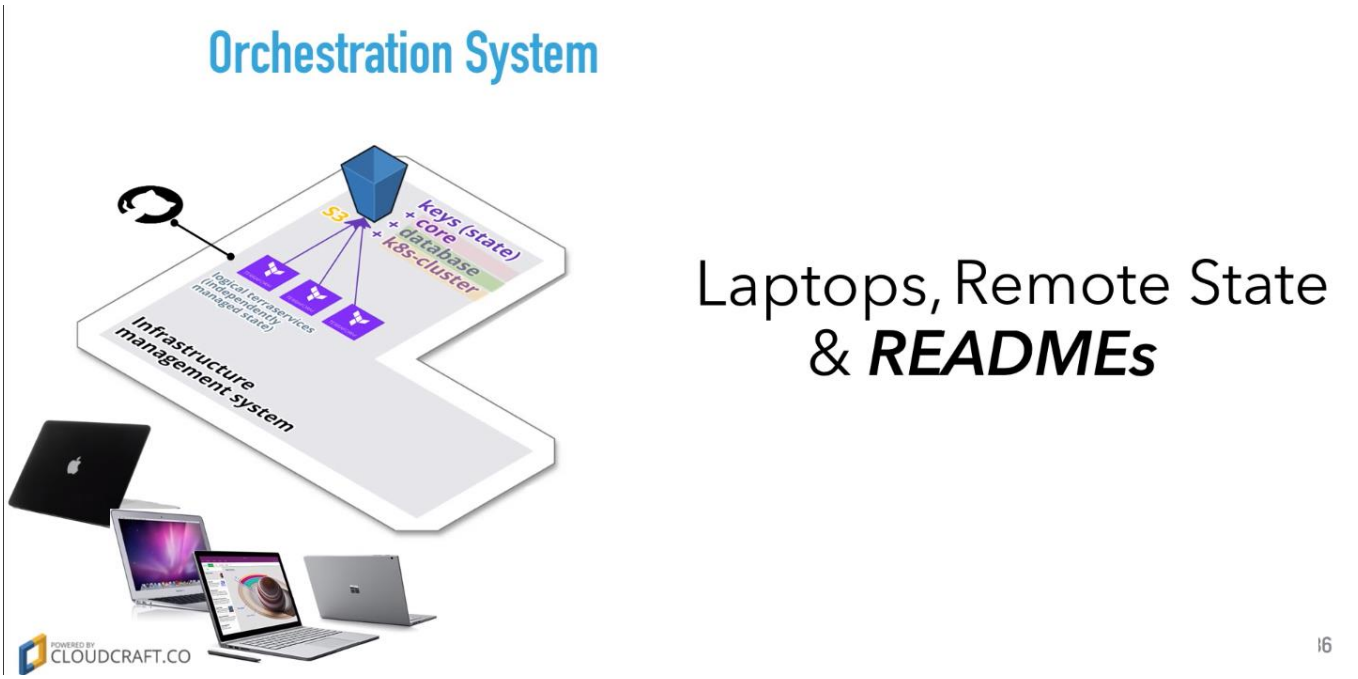


We started with a single developer using a single laptop with a single GIT repo and a single state environment file

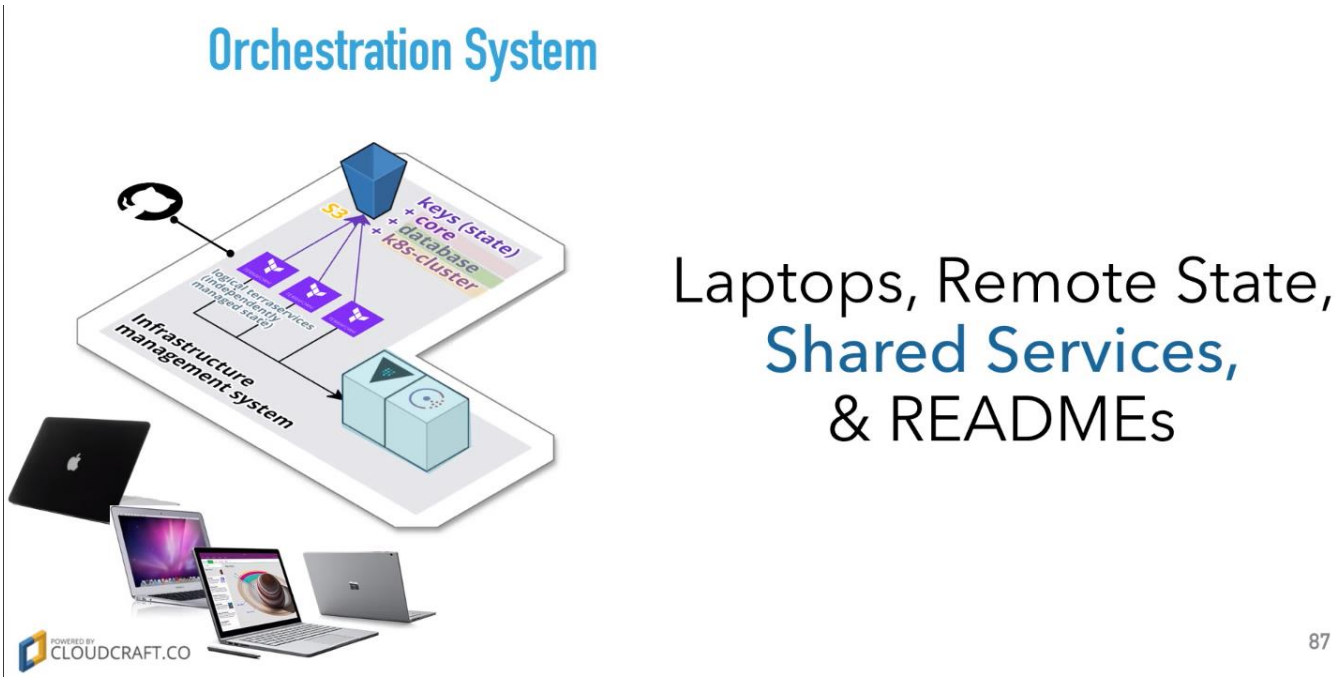
Then we have more developers that are all trying to do things concurrently, and things get complicated and we need some coordination



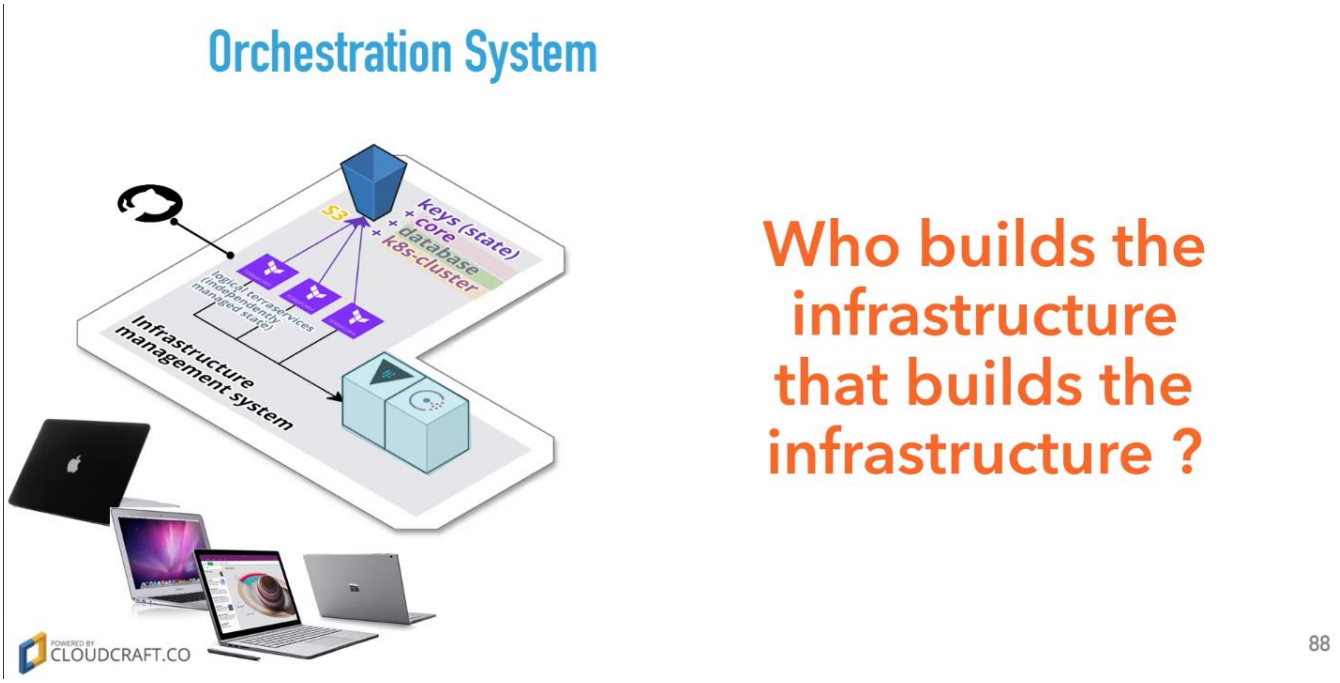
We can move to start using S3 as a store for the state files that hold our module configurations

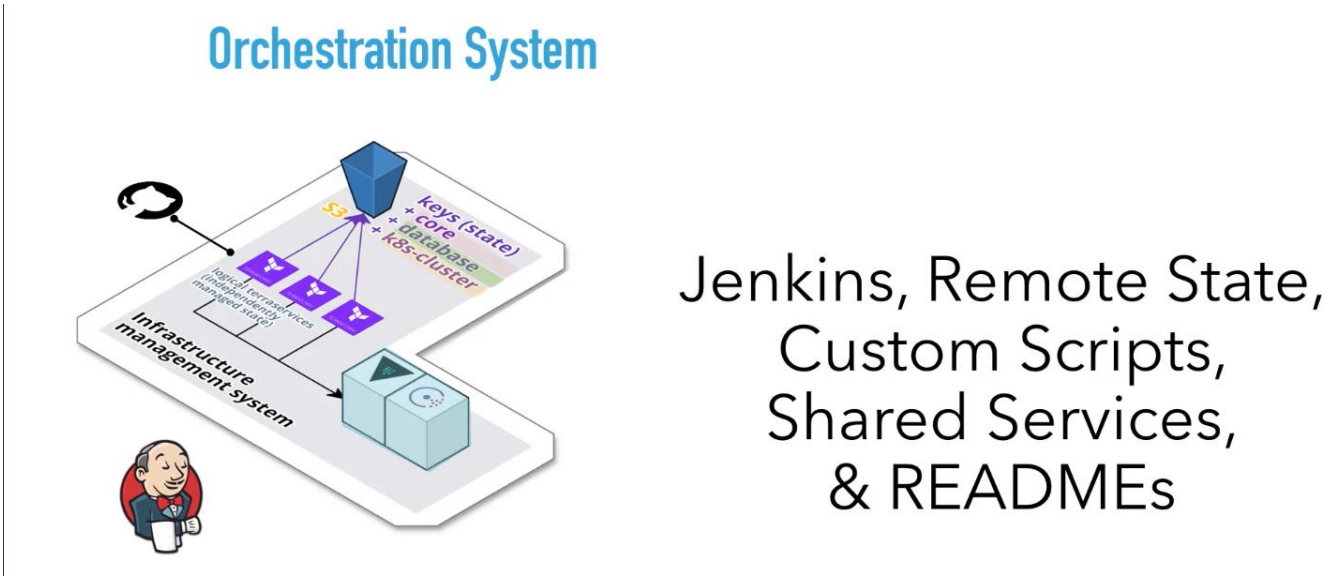


As the multiple state files start getting complicated too, we need some more communication between team members

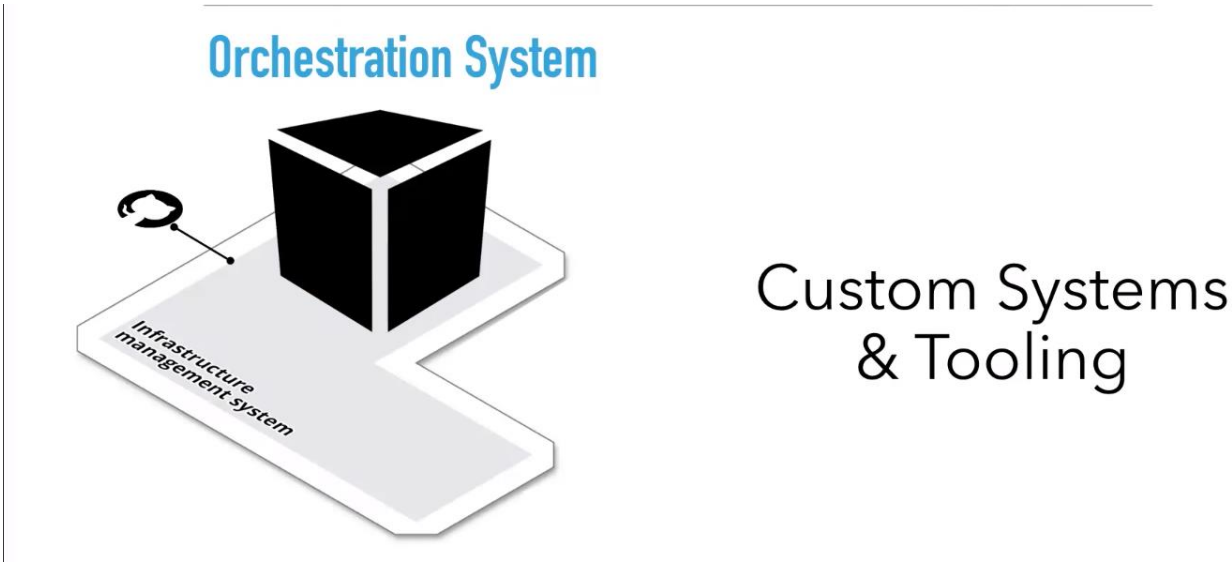


People also start using some provisioning tool like Ansible or puppet for installing things on the instances as well. You can use things like Consul or Vault clusters for holding shared data between Terraform and Ansible

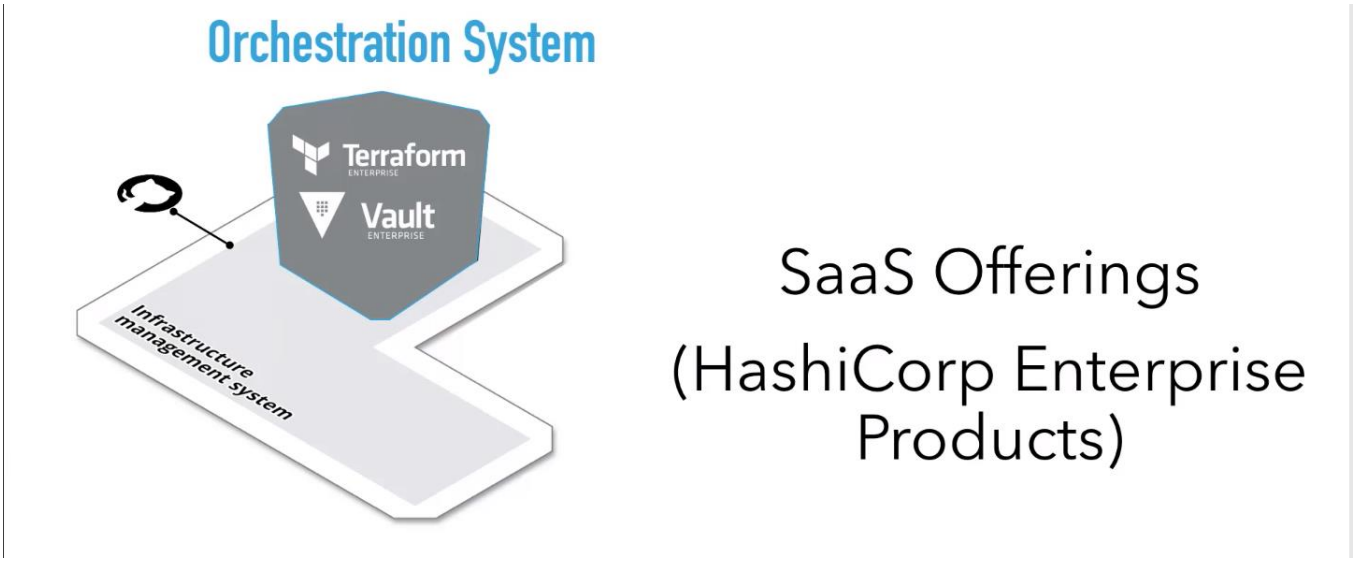




As a start, people can start using Jenkins for running Terraform scripts



Many clients also end up writing their own custom Terraform tooling like TerraGrunt, TerraHelp, etc.



It's not just about the structure of the code …

You also need to evolve your supporting orchestration system & processes

Conclusion

Evolving Terraform Setup

EVOLVING YOUR TERRAFORM SETUP

- Terralith
- Multi Terralith - Envs: Independent management
- Terramod
- Terramod^{n} - Modules : Maintainability & Reuse
- Terraservices - Logical Components: Independent management

Also need to consider how to evolve the management & orchestration of Terraform