4 days training

- Day 1 - Ansible
  - Ansible deep dive
  - Ansible on Azure
  - Ansible tower
- Day 2
  - Terraform deep-dive
  - Terraform on Azure
  - Terraform Enterprise
- Day 3
  - Puppet core
  - Puppet enterprise modules
- Day 4
  - Puppet bolt
  - Comparison
  - Enterprise level setup, troubleshoot, pipeline , access level
  - Nagios
  - Selenium integration in pipeline

Environment setup

Vs code plugins

- ansible
- yaml

Exercise 1

Write playbook to install and run apache

Exercise 2

Add handler to playbook to install and run apache

Exercise 3

Create template to update dynamic content

Exercise 4

1. Download roles from Galaxy
2. Create Ansible role to install apache

Exercise 5

Create Ansible vault

Exercise 6

Write Azure ansible module to provision VM in Azure

Exercise 6

Write Azure ansible playbook to provision VM in Azure

Exercise 7

Write Azure ansible playbook to provision Azure SQL

Exercise 8

Setup Tower

tar xvzf ansible-tower-setup-latest.tar.gz

./setup.sh

```
[tower]

localhost ansible_connection=local

[database]

[all:vars]

admin_password='password'

pg_host=''

pg_port=''

pg_database='awx'

pg_username='awx'

  pg_password='password'
```

Topics to be covered

Security - Done, Provision new azure resources without ansible az modules - https://docs.ansible.com/ansible/latest/modules/azure_rm_deployment_module.html

Extendability of existing modules, Creating custom modules - https://d2c.io/post/extending-ansible-modules

Private galaxy role - git is option https://www.reddit.com/r/ansible/comments/ctuj7h/private_ansible_galaxy/ Done

Custom RBAC - - https://docs.ansible.com/ansible-tower/3.7.2/html/userguide/security.html#rbac-ug Done

Webook services - https://docs.ansible.com/ansible-tower/latest/html/userguide/webhooks.html Done

https://www.ansible.com/blog/intro-to-automation-webhooks-for-red-hat-ansible-automation-platform

Topics to be covered

Backups and restore of Tower - https://docs.ansible.com/ansible-tower/latest/html/administration/backup_restore.html#ag-clustering-backup-restore

Sample code to show private github repo integration in playbook

Sample code to show custom module.Ansible command to execute parallely

Provisioning multiple Azure components example: **VM** using same playbook

Backup of Ansible core

Exercise 9 - 15 minutes

Complete below steps in tower

- Create Org1, Org2, each org add team1, team2, each team add user1, user2
- Give diff permission to org, team, users
- Create inventory
- Create Credential
- Create Project for https://github.com/ansible/tower-example playbook
- Create Template, Workflow template
- Run Job
- Customization

TERRAFORM

Exercise 10 - Install terraform

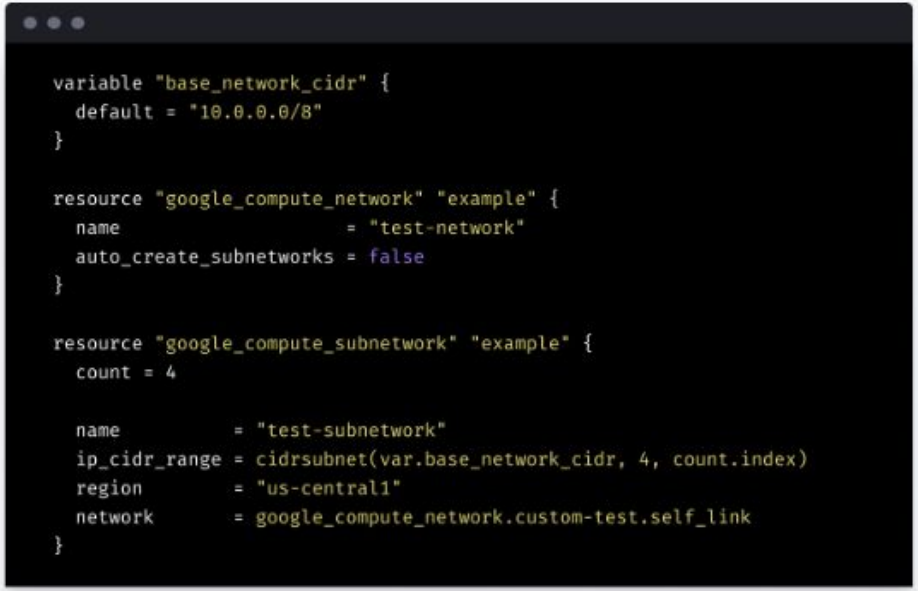
- sudo yum install -y yum-utils
- sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/RHEL/hashicorp.repo
- sudo yum -y install terraform
- terraform -help
- terraform -help plan

Deliver infrastructure as code with Terraform

Write declarative configuration files

- Collaborate and share configurations
- Evolve and version your infrastructure
- Automate provisioning

Define infrastructure as code to manage the full lifecycle — create new resources, manage existing ones, and destroy those no longer needed.

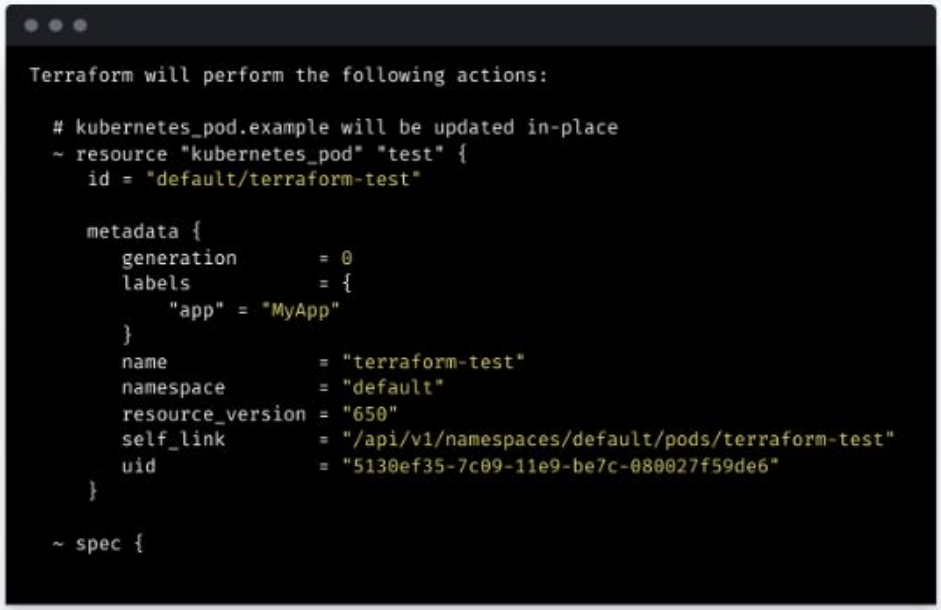


```
variable "base_network_cidr" {
  default = "10.0.0.0/8"
}

resource "google_compute_network" "example" {
  name                    = "test-network"
  auto_create_subnetworks = false
}

resource "google_compute_subnetwork" "example" {
  count = 4

  name          = "test-subnetwork"
  ip_cidr_range = cidrsubnet(var.base_network_cidr, 4, count.index)
  region        = "us-central1"
  network       = google_compute_network.custom-test.self_link
}
```



Plan and predict changes

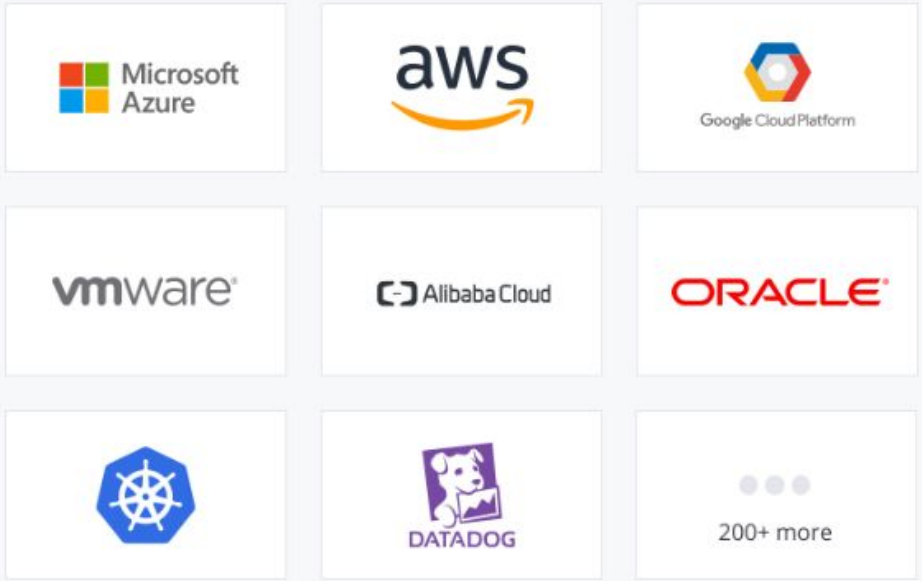
- Clearly mapped resource dependencies
- Separation of plan and apply
- Consistent, repeatable workflow

Terraform provides an elegant user experience for operators to safely and predictably make changes to infrastructure.

Create reproducible infrastructure

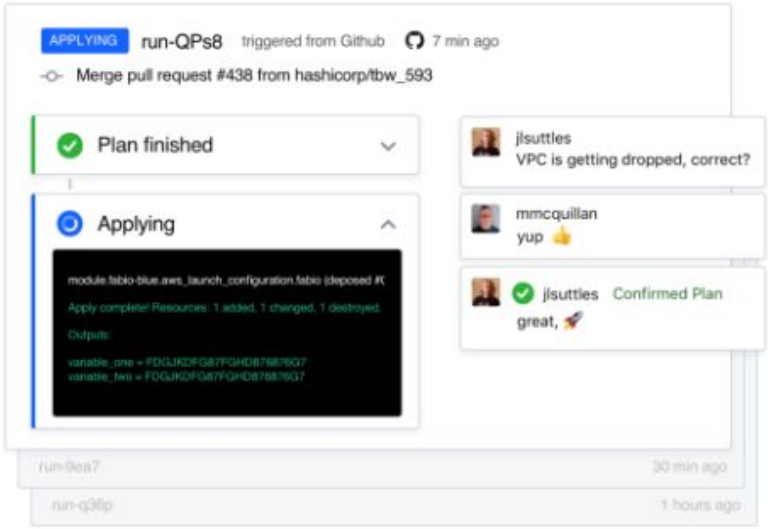
- Reproducible production, staging, and development environments
- Shared modules for common infrastructure patterns
- Combine multiple providers consistently

Terraform makes it easy to re-use configurations for similar infrastructure, helping you avoid mistakes and save time.



Enhanced Workflow for Teams with Terraform Cloud

Terraform Cloud is a free to use SaaS application that provides the best workflow for writing and building infrastructure as code with Terraform.



Share infrastructure as code

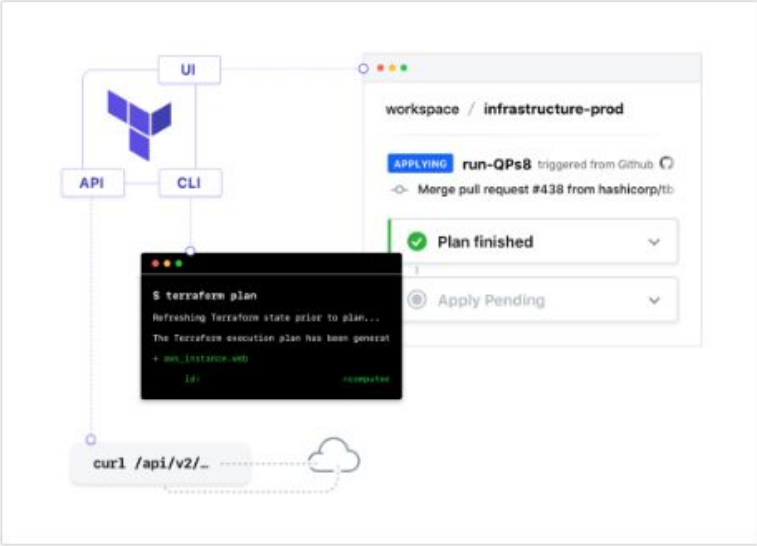
Empower your team to rapidly review, comment, and iterate on Infrastructure as Code.

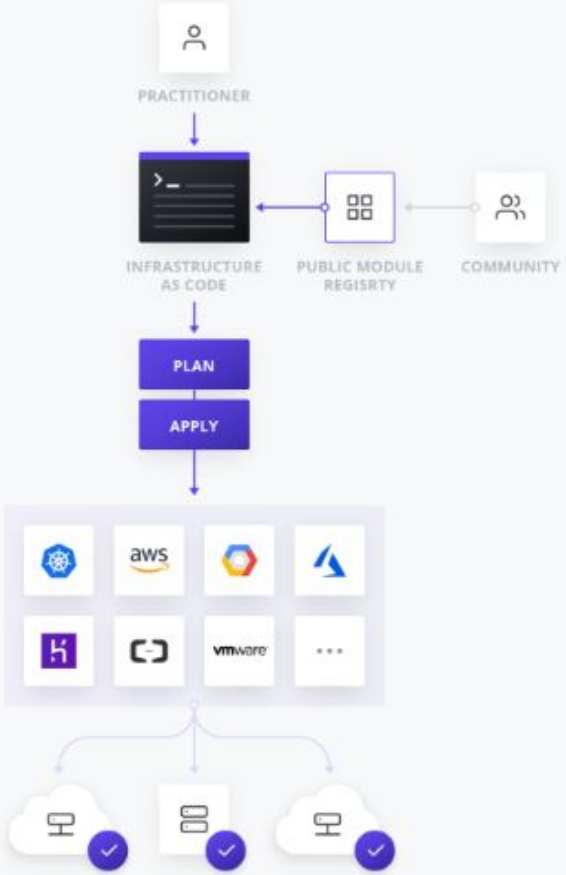
- State management (storage, viewing, history, and locking)
- Web UI for viewing and approving Terraform runs
- Collaborative Runs
- Private module registry

Automate consistent workflows

Create a pipeline for provisioning Infrastructure as Code

- VCS integration (Azure DevOps, Bitbucket, GitHub, and GitLab)
- Enable GitOps workflow
- Remote operations — perform Terraform runs in a consistent, immutable environment
- Notifications for run events (via Slack or webhooks)
- Full HTTP API for integrating with other tools and services

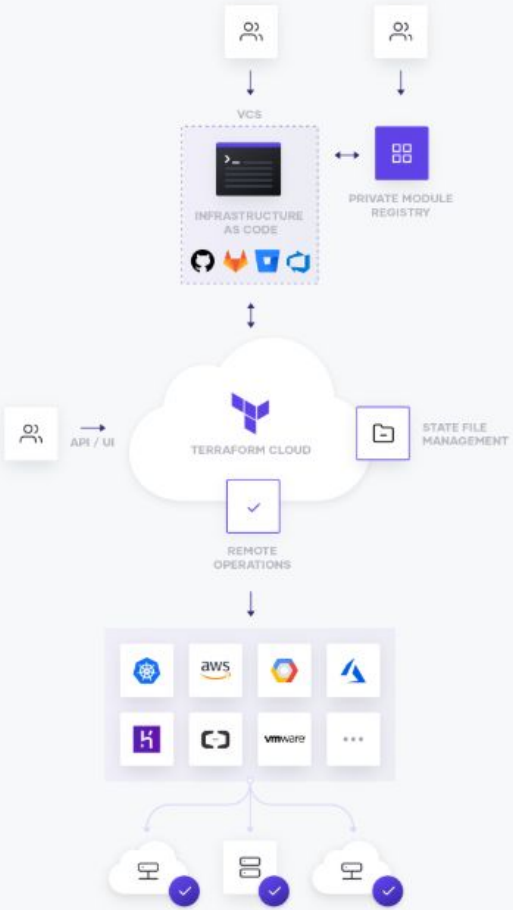




CLI

Terraform allows infrastructure to be expressed as code in a simple, human readable language called HCL (HashiCorp Configuration Language). Terraform CLI reads configuration files and provides an execution plan of changes, which can be reviewed for safety and then applied and provisioned. Extensible providers allow Terraform to manage a broad range of resources, including hardware, IaaS, PaaS, and SaaS services.

- Infrastructure as Code
- 200+ available providers
- Provision any infrastructure



Terraform Cloud

Write Infrastructure as Code

Terraform users define infrastructure in a simple, human-readable configuration language called HCL (HashiCorp Configuration Language). Users can write unique HCL configuration files or borrow existing templates from the public module registry.

Manage Configuration Files in VCS

Most users will store their configuration files in a version control system (VCS) repository and connect that repository to a Terraform Cloud workspace. With that connection in place, users can borrow best practices from software engineering to version and iterate on infrastructure as code, using VCS and Terraform Cloud as a delivery pipeline for infrastructure.

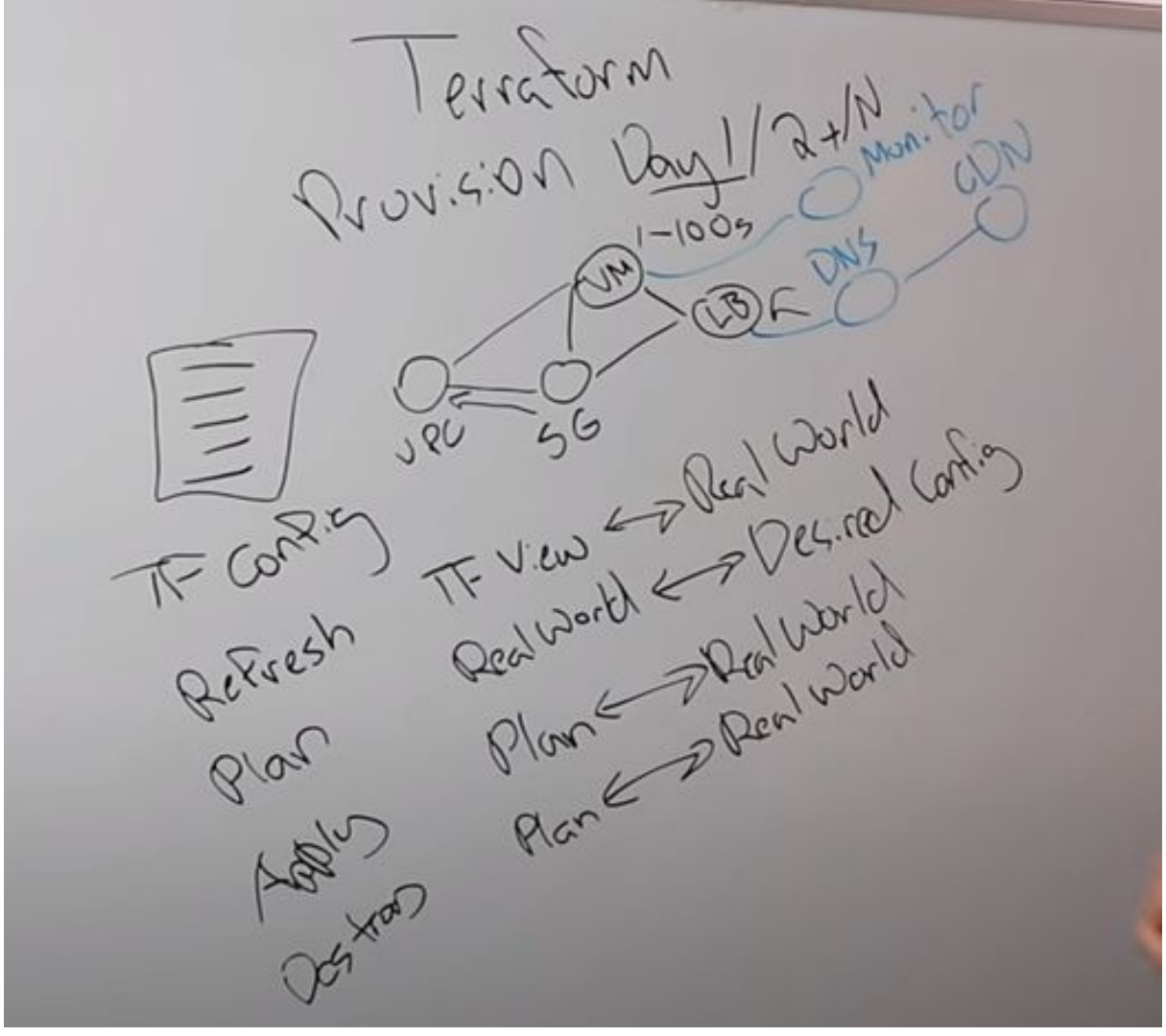
Automate Provisioning

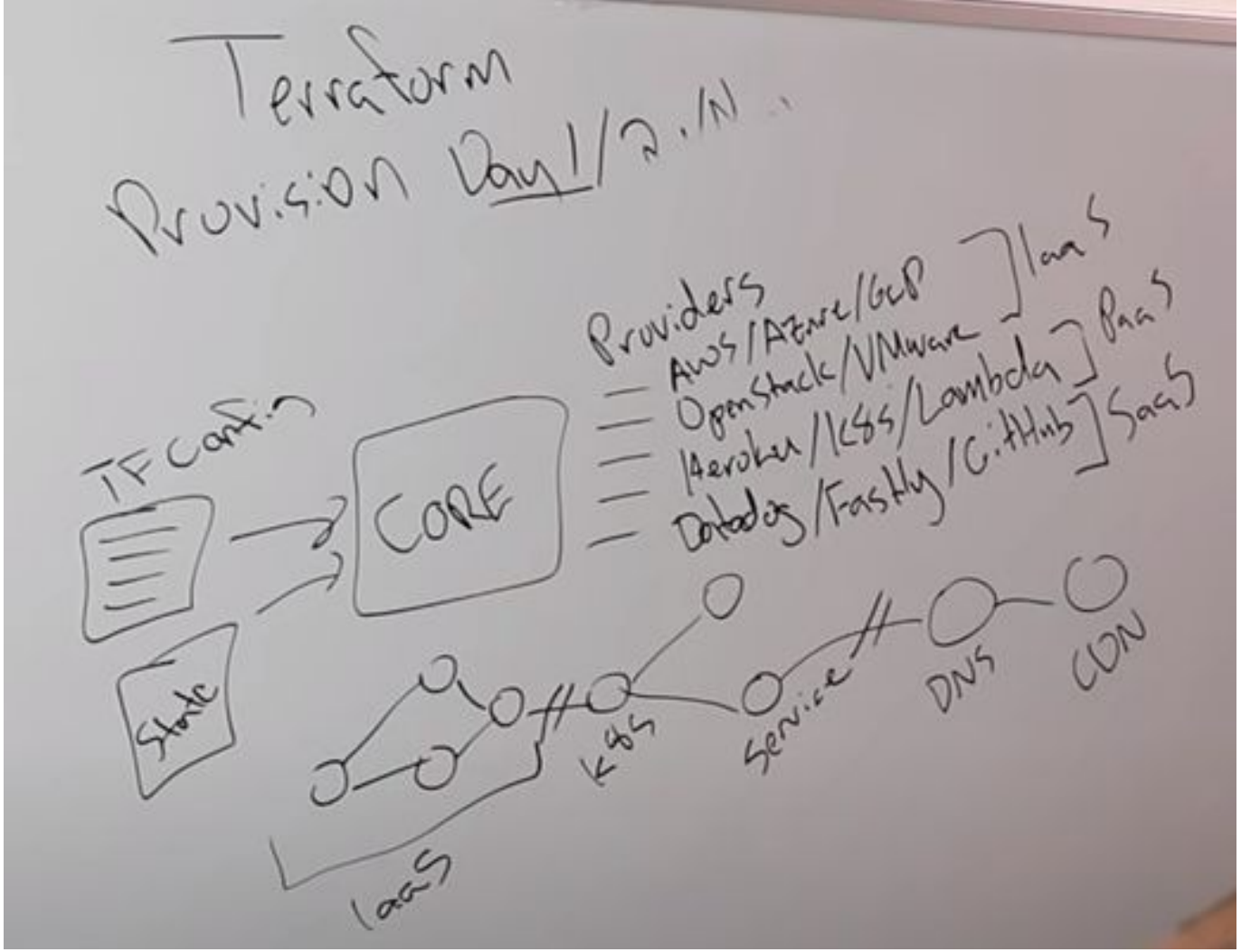
When you push changes to a connected VCS repository, Terraform Cloud will automatically trigger a plan in any workspace connected to that repository. This plan can be reviewed for safety and accuracy in the Terraform UI, then it can be applied to provision the specified infrastructure.

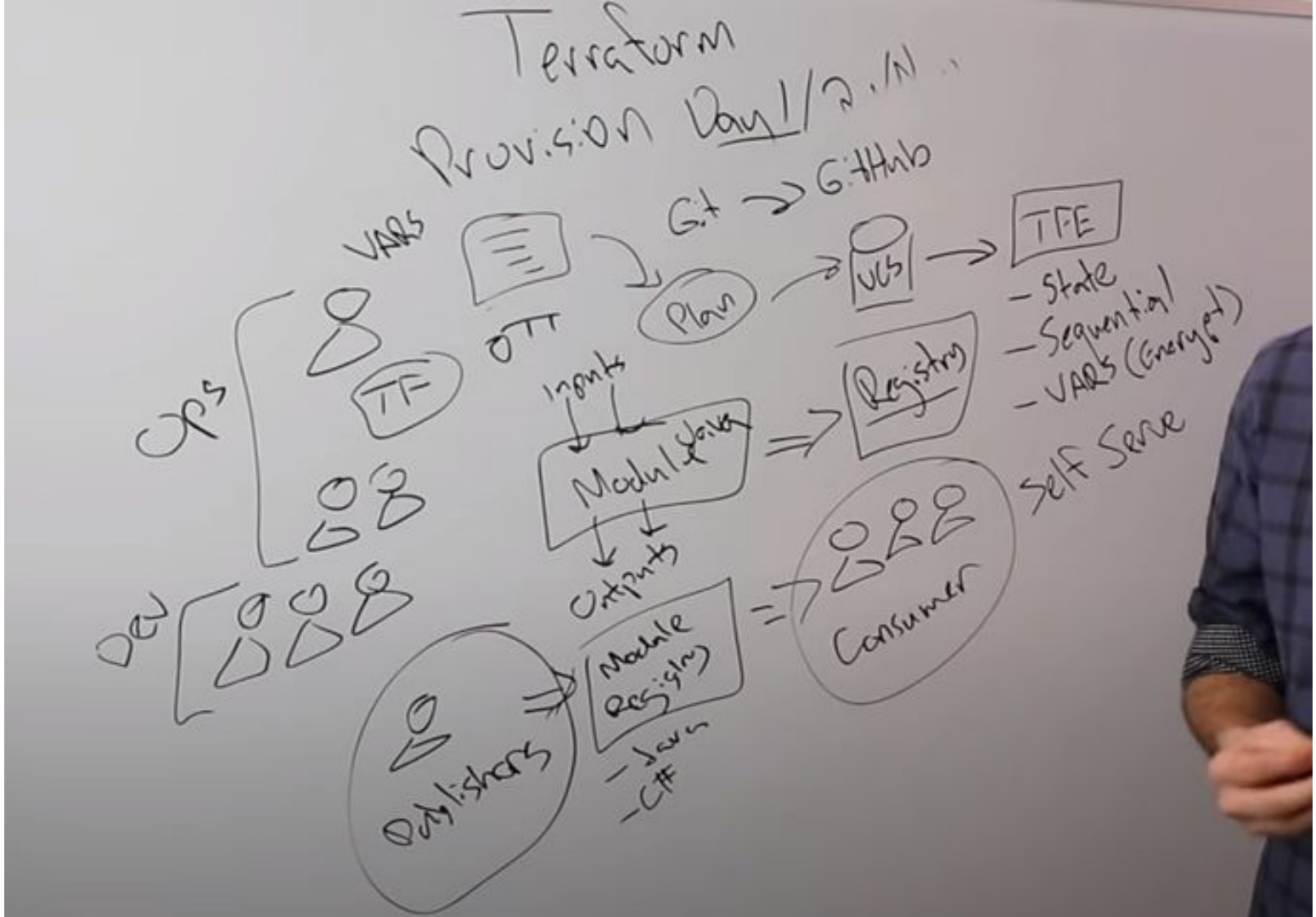
Strong Community

- 25,000+ Commits
- 1,000+ Modules
- 200+ Providers

Open Source projects benefit from the scrutiny of a broad and diverse user base. Keeping the code available helps to teach and empower the community of users, while it also provides an easy mechanism for feedback, improvement, and customization.


Terraform
Provision Day 1 / 2+/N
Monitor
CDN
1-100s
VM
LB
DNS
VPC
SG
TF Config
Refresh
Plan
Apply
Destroy
TF View ↔ Real World
Real World ↔ Desired Config
Plan ↔ Real World
Plan ↔ Real World


Terraform
Provision Day1/2.IN
Providers
AWS/Azure/GCP
OpenStack/VMware
Heroku/CF/Lambda
Datadog/Fastly/GitHub
IaaS
PaaS
SaaS
TF Config
CORE
State
K8S
service
DNS
CDN
IaaS


Terraform
Provision Day 1/2 ...
VARS
TF
OTT
Git → GitHub
Plan
VCS
TFE
- State
- Sequential
- VARS (Encrypted)
Ops
Dev
Inputs
Module
Registry
Outputs
Module Registry
- Java
- C#
Publishers
Consumer
Self Serve

Exercise 11

Provision AWS components

- Install AWS cli
  - curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
  - unzip awscliv2.zip
  - sudo ./aws/install
  - aws --version
- Create Key in AWS
- Configure credentials - aws configure

Exercise 12

Provision Docker container with Terraform

- **Install Docker container**

  https://docs.docker.com/engine/install/centos/

Exercise 13

Provision Azure storage

- https://docs.microsoft.com/en-us/cli/azure/install-azure-cli-yum?view=azure-cli-latest

Exercise 14

Terraform Cloud/Enterprise

https://hashicorp.github.io/field-workshops-terraform/slides/aws/terraform-cloud/#1

https://play.instruqt.com/hashicorp/tracks/vault-basics/challenges/vault-cli/notes

UI Integration with puppet core https://www.theforeman.org/

Canary deployment terraform, rollback, rollforwards

Terraform one code to multi cloud

Software Provisioning with Provisioners

Software Provisioning with Ansible

Interpolation Expressions

Locals

Data Sources

Modules

Backends and Remote State

Workspaces

Interpolation Expressions

Expressions are used to refer to or compute values within a configuration. The simplest expressions are just literal values, like `"hello"` or `5`, but the Terraform language also allows more complex expressions such as references to data exported by resources, arithmetic, conditional evaluation, and a number of built-in functions.

Expressions can be used in a number of places in the Terraform language, but some contexts limit which expression constructs are allowed

```
{
  name = "John"
  age  = 52
}
```

Types and Values

The result of an expression is a *value*. All values have a *type*, which dictates where that value can be used and what transformations can be applied to it.

The Terraform language uses the following types for its values:

- `string`: a sequence of Unicode characters representing some text, like `"hello"`.
- `number`: a numeric value. The `number` type can represent both whole numbers like `15` and fractional values like `6.283185`.
- `bool`: either `true` or `false`. `bool` values can be used in conditional logic.
- `list` (or `tuple`): a sequence of values, like `["us-west-1a", "us-west-1c"]`. Elements in a list or tuple are identified by consecutive whole numbers, starting with zero.
- `map` (or `object`): a group of values identified by named labels, like `{name = "Mabel", age = 52}`.

Type Conversion

Expressions are most often used to set values for the arguments of resources and child modules. In these cases, the argument has an expected type and the given expression must produce a value of that type.

Where possible, Terraform automatically converts values from one type to another in order to produce the expected type. If this isn't possible, Terraform will produce a type mismatch error and you must update the configuration with a more suitable expression.

Terraform automatically converts number and bool values to strings when needed. It also converts strings to numbers or bools, as long as the string contains a valid representation of a number or bool value.

- `true` converts to `"true"`, and vice-versa
- `false` converts to `"false"`, and vice-versa
- `15` converts to `"15"`, and vice-versa

Locals values

A local value assigns a name to an expression, allowing it to be used multiple times within a module without repeating it.

Comparing modules to functions in a traditional programming language: if input variables are analogous to function arguments and outputs values are analogous to function return values, then *local values* are comparable to a function's local temporary symbols.

A set of related local values can be declared together in a single `locals` block:

```
locals {
  service_name = "forum"
  owner        = "Community Team"
}
```

The expressions assigned to local value names can either be simple constants like the above, allowing these values to be defined only once but used many times, or they can be more complex expressions that transform or combine values from elsewhere in the module:

```
locals {
  # Ids for multiple sets of EC2 instances, merged together
  instance_ids = concat(aws_instance.blue.*.id, aws_instance.green.*.id)
}

locals {
  # Common tags to be assigned to all resources
  common_tags = {
    Service = local.service_name
    Owner   = local.owner
  }
}
```

As shown above, local values can be referenced from elsewhere in the module with an expression like `local.common_tags`, and locals can reference each other in order to build more complex values from simpler ones.

```
resource "aws_instance" "example" {
  # ...

  tags = local.common_tags
}
```

Data Sources

Data sources allow data to be fetched or computed for use elsewhere in Terraform configuration. Use of data sources allows a Terraform configuration to make use of information defined outside of Terraform, or defined by another separate Terraform configuration.

Using Data Sources

A data source is accessed via a special kind of resource known as a *data resource*, declared using a `data` block:

```
data "aws_ami" "example" {
  most_recent = true

  owners = ["self"]
  tags = {
    Name   = "app-server"
    Tested = "true"
  }
}
```

```
# Find the latest available AMI that is tagged with Component = web
data "aws_ami" "web" {
  filter {
    name   = "state"
    values = ["available"]
  }

  filter {
    name   = "tag:Component"
    values = ["web"]
  }

  most_recent = true
}
```

Modules

A *module* is a container for multiple resources that are used together. Modules can be used to create lightweight abstractions, so that you can describe your infrastructure in terms of its architecture, rather than directly in terms of physical objects.

The `.tf` files in your working directory when you run `terraform plan` or `terraform apply` together form the *root* module. That module may call other modules and connect them together by passing output values from one to input values of another.

To learn how to *use* modules, see the Modules configuration section. This section is about *creating* re-usable modules that other configurations can include using `module` blocks.

You can also learn more about how to use and create modules with our hands-on modules track on learn.hashicorp.com.

```
$ tree minimal-module/
.
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
```

A complete example of a module following the standard structure is shown below. This example includes all optional elements and is therefore the most complex a module can become:

```
$ tree complete-module/
.
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
├── ...
├── modules/
│   ├── nestedA/
│   │   ├── README.md
│   │   ├── variables.tf
│   │   ├── main.tf
│   │   ├── outputs.tf
│   ├── nestedB/
│   ├── .../
├── examples/
│   ├── exampleA/
│   │   ├── main.tf
│   ├── exampleB/
│   ├── .../
```

Publishing Modules

If you've built a module that you intend to be reused, we recommend publishing the module on the Terraform Registry. This will version your module, generate documentation, and more.

Published modules can be easily consumed by Terraform, and users can constrain module versions for safe and predictable updates. The following example shows how a caller might use a module from the Terraform Registry:

```
module "consul" {
  source = "hashicorp/consul/aws"
}
```

If you do not wish to publish your modules in the public registry, you can instead use a private registry to get the same benefits.