Terraform Core & Beyond

TEKsystems Global Services

WORKFORCE DEVELOPMENT

Logistics

- Class Hours:
- Instructor will provide class start and end times.
- Breaks throughout class

- Telecommunication:
- Turn off or set electronic devices to vibrate
- Reading or attending to devices can be distracting to other students

- Lunch:
- 1 hour 15 minutes

- Miscellaneous
- Courseware
- Bathroom

Course Objectives

- Explain Infrastructure-as-Code and some of the tools for implementing it.
- Describe what Configuration Drift is and how to avoid it.
- Understand the benefits of using Terraform
- Manage Terraform State
- Learn best practices for building secure Terraform configurations.
- Use Terraform to automate infrastructure management.
- Terraform Testing
- More!

Introductions

- Name
- Job Role
- Your experience with Scripting (scale 1-5)
- Your experience with Python (scale 1 - 5)
- Your experience with Ansible
- Your experience with Terraform
- Expectations for the course (please be specific)

Lab page

https://github.com/jrues/tf-core

Configuration Management

Configuration Drift

POTHOLE

- Your infrastructure requirements change
- Configuration of a server falls out of policy

- Manage with Infrastructure as Code (IAC)
  - Terraform
  - Ansible
  - Chef
  - Puppet

Manual

Common manual tasks

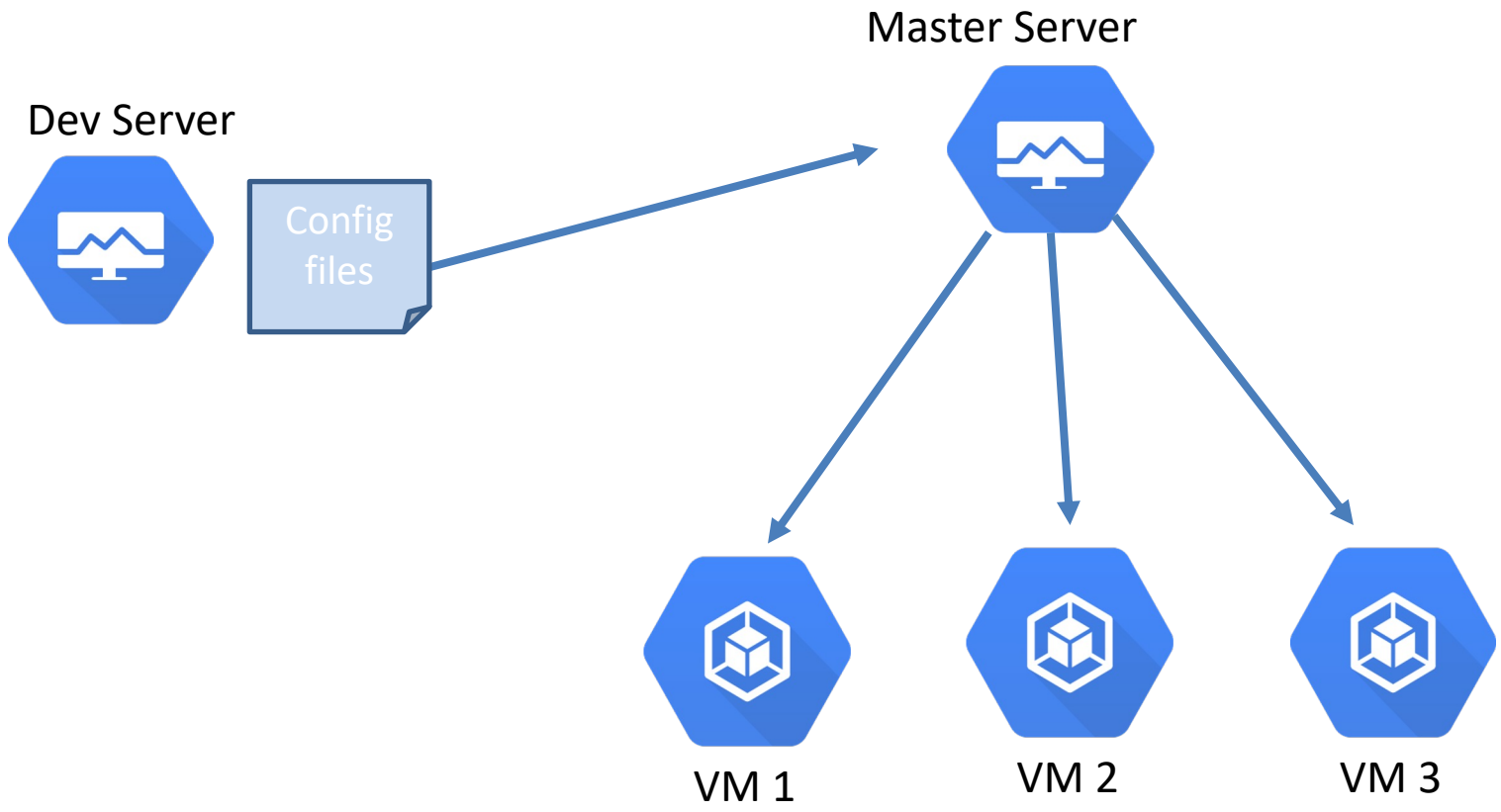
Installation and configuration:

- Operating System
- Runtime
- Libraries
- Utilities
- Configuration files

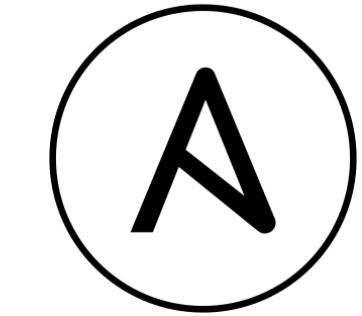
- Build application
- Test application
- Deploy application

Automated

Automated



Automation tools



ANSIBLE

CHEF

puppet

HashiCorp Terraform

Infrastructure as Code

What is IaC?

Infrastructure as code (IaC) is the process of managing and provisioning computer data centers through machine-readable definition files
Used with bare-metal as well as virtual machines and many other resources.
Normally, a declarative approach

Infrastructure as Code

- Programmatically provision and configure components
- Treat like any other code base
  - Version control
  - Automated testing
  - data backup

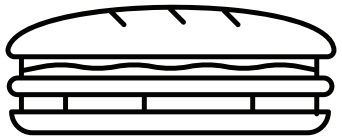
Infrastructure as Code

Provisioning infrastructure through software to achieve consistent and predictable environments.

Core Concepts

- Defined in code
- Stored in source control
- Declarative or imperative

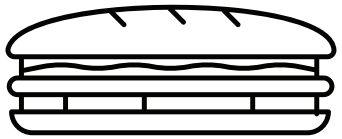
Imperative



```
# Make a sandwich
get bread
get mayo
get turkey
get lettuce

spread mayo on bread
put lettuce in between bread
put turkey in between bread
on top of turkey
```

Declarative



```
# Make a sandwich
food sandwich "turkey" {
  ingredients = [
    "bread", "turkey",
"mayo", "lettuce"
  ]
}
```

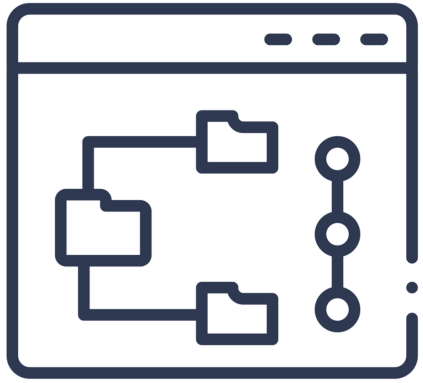
POP QUIZ:

What challenges does Infrastructure as Code solve?

Terraform Introduction

Automating Infrastructure

Provisioning resources

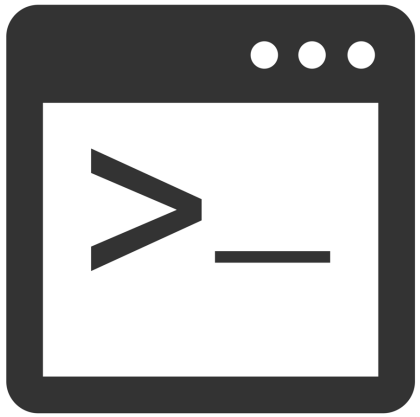


Version Control

Plan Updates

Reusable Templates

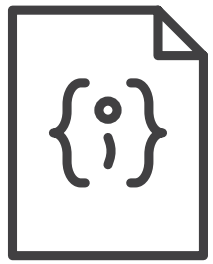
Terraform components



Terraform executable

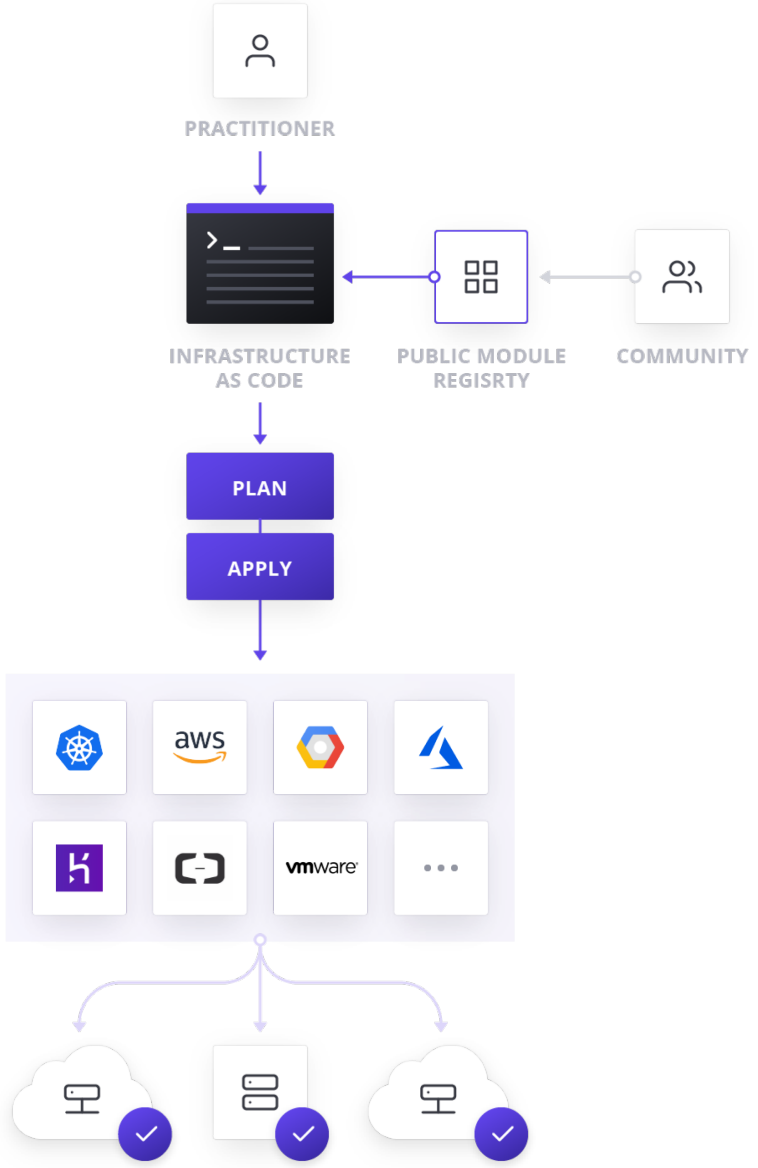
Terraform files

Terraform plugins



Terraform state

Terraform architecture



POP QUIZ:

What is Terraform used for?

Terraform CLI

Run `terraform` command

Output:

```
Usage: terraform [-version] [-help] <command> [args]

The available commands for execution are listed below.
The most common, useful commands are shown first, followed by
less common or more advanced commands. If you're just getting
started with Terraform, stick with the common commands. For the
other commands, please read the help and docs before usage.

Common commands:
    apply              Builds or changes infrastructure
    console            Interactive console for Terraform
interpolations
    destroy            Destroy Terraform-managed infrastructure
    env                Workspace management
    fmt                Rewrites config files to canonical format
    get                Download and install modules for the
configuration
    graph              Create a visual graph of Terraform
resources
```

Terraform CLI

Command:

```
terraform init
```

Output:

```
Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "docker"...
```

Terraform fetches any required providers and modules and stores them in the .terraform directory. Check it out and you'll see a plugins folder.

Terraform CLI

Command:

```
terraform validate
```

Validate all of the Terraform files in the current directory. Validation includes basic syntax check as well as all variables declared in the configuration are specified.

Terraform CLI

Command:

```
terraform plan
```

Output:

```
Terraform will perform the following actions:

  + aws_instance.aws-k8s-master
      id:                                   <computed>
      ami:                                  "ami-01b45..."
      instance_type:                        "t3.small"
```

Plan is used to show what Terraform will do if applied. It is a dry run and does not make any changes.

Terraform CLI

Command:

```
terraform apply
```

Output:

```
❯ terraform apply "rapid-app.out"
aws_security_group.k8s_sg: Creating...
  arn:                                      "" => "<computed>"
  description:                              "" => "Allow all
inbound traffic necessary for k8s"
  egress.#:                                 "" => "1"
  egress.482069346.cidr_blocks.#:           "" => "1"
  egress.482069346.cidr_blocks.0:           "" => "0.0.0.0/0"
```

Performs the actions defined in the plan

Terraform CLI

Command:

```
terraform destroy [-auto-approve]
```

Destroys all the resources in the state file.
-auto-approve (don't prompt for confirmation)

Lab: Setup VM

Lab: Build first instance

Terraform CLI

Terraform is normally run from inside the directory containing the *.tf files for the root module. Terraform checks that directory and automatically executes them.

In some cases, it makes sense to run the Terraform commands from a different directory. This is true when wrapping Terraform with automation. To support that Terraform can use the global option `-chdir=...` which can be included before the name of the subcommand.

Terraform CLI

The `chdir` option instructs Terraform to change its working directory to the given directory before running the given subcommand. This means that any files that Terraform would normally read or write in the current working directory will be read or written in the given directory instead.

Terraform CLI

Command:

```
terraform -chdir=environments/dev (apply|plan|destroy)
```

Output:

```
❯ terraform apply
aws_security_group.k8s_sg: Creating...
  arn:                                      "" => "<computed>"
  description:                              "" => "Allow all
inbound traffic necessary for k8s"
  egress.#:                                 "" => "1"
  egress.482069346.cidr_blocks.#:           "" => "1"
  egress.482069346.cidr_blocks.0:           "" => "0.0.0.0/0"
```

Performs the subcommand in the specified directory.

Terraform CLI

It can be time consuming to update a configuration file and run `terraform apply` repeatedly to troubleshoot expressions not working.

Terraform has a console subcommand that provides an interactive console for evaluating these expressions.

Terraform CLI

```
variable "x" {

default = [
        {
          name = "first",
          condition = {
            age = "1"
          }
          action = {
            type = "Delete"
          }
        },{
          name = "second",
          condition = {
            age = "2"
          }
          action = {
            type = "Delete"
          }
        }
    ]
}
```

Terraform CLI

Command:

```
terraform console
```

Output:

```
> var.x[1].name
"second"

var.x[0].condition
{
  "age" = "1"
}
```

Test expressions and interpolations interactively.

Terraform vs JSON

ARM JSON:

```
"name" : "[concat(parameters('PilotServerName'), '3')]",
```

Terraform:

```
name = "${var.PilotServerName}3"
```

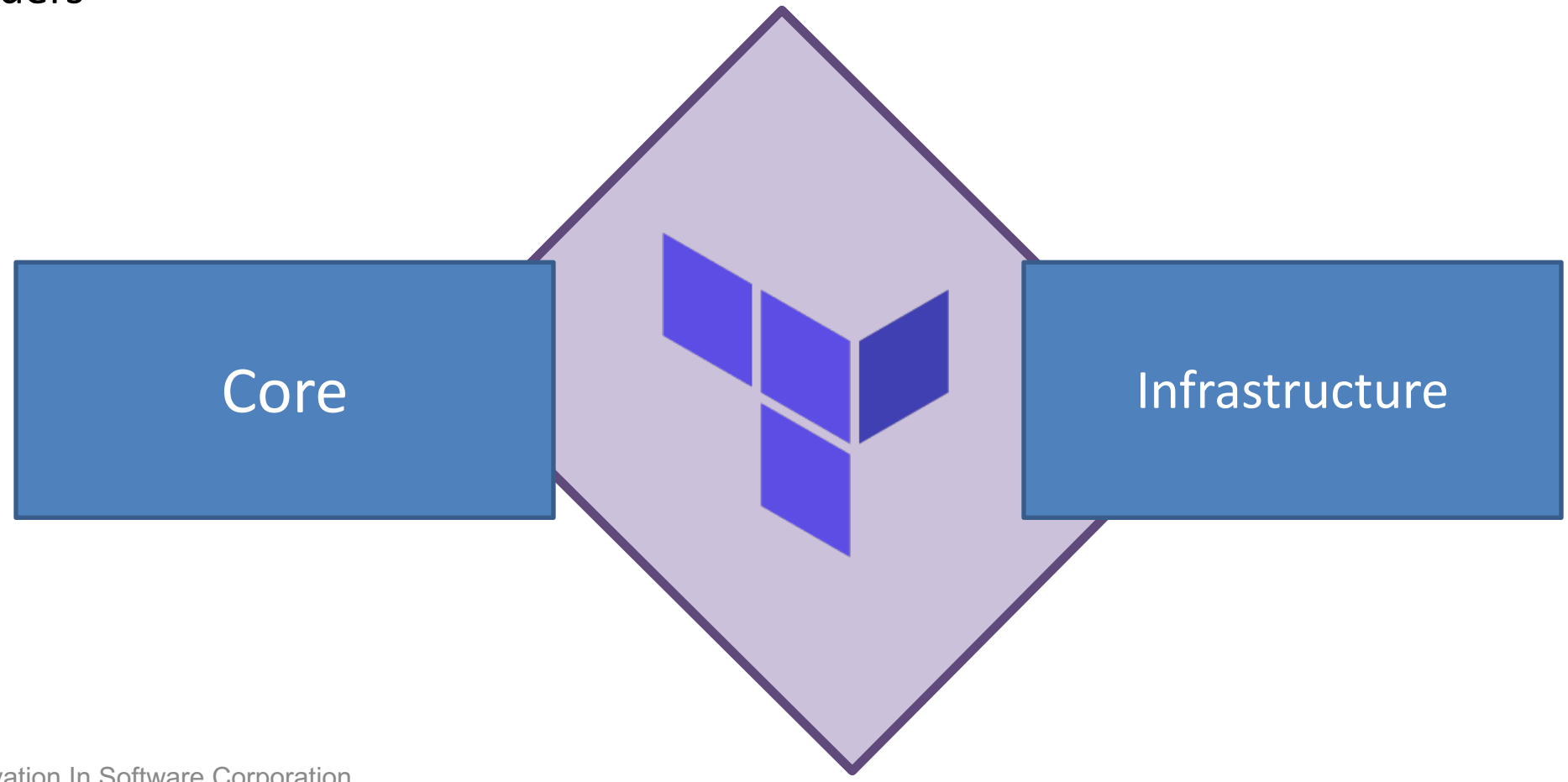
Terraform code (HCL) is "easy" to learn and "easy" to read. It is also more compact than equivalent JSON configuration.

Terraform model

Terraform code is broken into three parts: Core Terraform, Modules and Providers

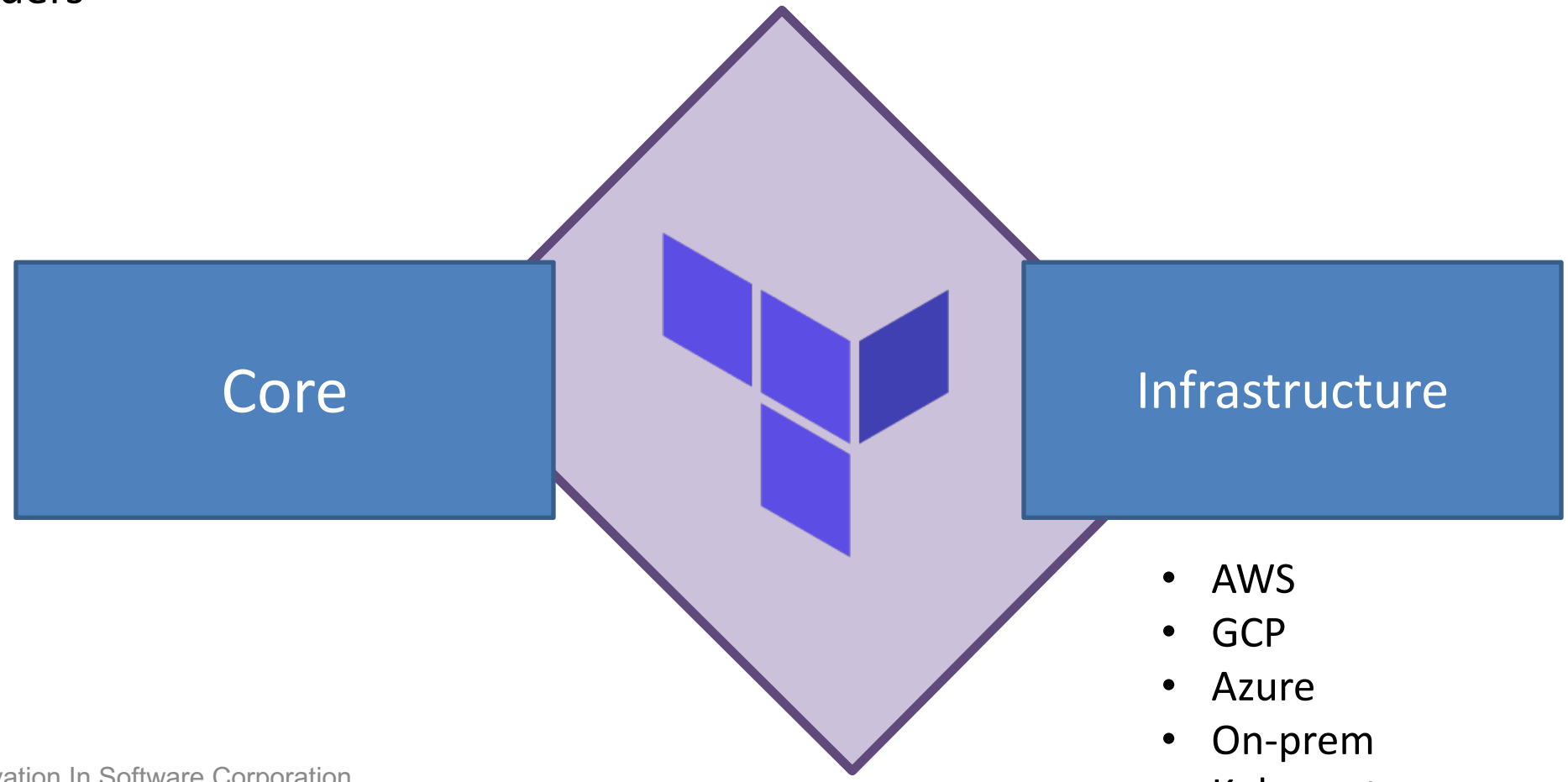
Terraform model

Terraform code is broken into three parts: Core Terraform, Modules and Providers



Terraform model

Terraform code is broken into three parts: Core Terraform, Modules and Providers



- AWS
- GCP
- Azure
- On-prem
- Kubernetes

Configuration files

Terraform has a pretty confusing file hierarchy. We are going to start with the basics.

`provider.tf` - specify provider information

`main.tf` - Create infrastructure resources.

`variables.tf` - define values for variables in main.tf

Terraform files

- core = Terraform language, logic, and tooling.
- provider = Pluggable code to allow Terraform to talk to vendor APIs
- modules = A container for multiple resources that are used together.

Terraform providers

Terraform relies on plugins called "providers" to interface with remote systems.

Terraform configurations must declare which providers they require so that Terraform can install and use them. Additionally, some providers require configuration (like endpoint URLs or cloud regions) before they can be used.

Terraform providers

Each provider adds resources and/or data sources that Terraform manages.

Every resource type is implemented by a provider. Terraform can't manage any kind of infrastructure without providers.

Providers are used to configure infrastructure platforms (either cloud or self-hosted). Providers also offer utilities for tasks like generating random numbers for unique resource names.

Terraform files

```
##Amazon Infrastructure
provider "aws" {
  region = "${var.aws_region}"
}
```

- Provider defines what infrastructure Terraform will be managing
  - Pass variables to provider
    - Region, Flavor, Credentials

Terraform provider configuration

```
# The default provider configuration; resources that begin with `aws_` will use
# it as the default, and it can be referenced as `aws`.
provider "aws" {
  region = "us-east-1"
}

# Additional provider configuration for west coast region; resources can
# reference this as `aws.west`.
provider "aws" {
  alias  = "west"
  region = "us-west-2"
}
```

Terraform supports 'aliases' for multiple configurations for the same provider, and you can select which one to use on a per-resource or per-module basis.

Terraform provider configuration

```
provider "azurerm" {
  version = "=1.30.1"
}
```

Terraform allows you to configure the providers in code. configure specific versions or pull in latest.

Terraform provider configuration

For Azure that can be az login, Managed Identity, or environment variables.

```
az login
```

```
export ARM_TENANT_ID=
export ARM_SUBSCRIPTION_ID=
export ARM_CLIENT_ID=
export ARM_CLIENT_SECRET=
```

Terraform provider configuration

Terraform will use the native tool's method of authentication. Environment variables, shared credentials files or static credentials.

```
provider "aws" {}
```

```
export AWS_ACCESS_KEY_ID=
export AWS_SECRET_ACCESS_KEY=
```

Terraform provider configuration

Terraform will use the native tool's method of authentication. Environment variables, shared credentials files or static credentials.

```
provider "aws" {
  region                   = "us-west-2"
  shared_credentials_file  = "~/.aws/creds"
  profile                  = "customprofile"
}
```

Terraform ASA provider

```
provider "ciscoasa" {
  api_url           = "https://10.0.0.5"
  username          = "admin"
  password          = "YOUR SECRET PASSWORD"
  ssl_no_verify     = false
}
```

Providers allow you to authenticate in the code block. **This is a very BAD idea.**

Use environment variables:

`CISCOASA_USER`
`CISCOASA_PASSWORD`

Terraform Resources

Terraform resources

Resources are the most important element of the Terraform language. Each resource block describes one or more infrastructure objects, such as networks, instances, or higher-level objects such as DNS records.

Terraform resources

A resource block declares a resource of a given type ("aws_instance") with a given local name ("web"). The name is used to refer to this resource from elsewhere in the same Terraform module but has no significance outside that module's scope.

The resource type and name together serve as an identifier for a given resource and so must be unique within a module.

Terraform resources

Every Terraform resource is structured the exact same way.

```
resource "type" "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
```

resource = top level keyword

Terraform resources

Within the block body (between { and }) are the configuration arguments for the resource itself. Most arguments in this section depend on the resource type, and indeed in this example both ami and instance_type are arguments defined specifically for the aws_instance resource type.

Terraform resources

Every Terraform resource is structured the exact same way.

```
resource "type" "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
```

type = this is the name of the resource. The first part tells you which provider is belongs to. Example: `azurerm_virtual_machine`. This means the provider is Azure and the specific type of resources is a virtual machine.

Terraform resources

Every Terraform resource is structured the exact same way.

```
resource "type" "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
```

name = arbitrary name to refer to resource. Used internally by TF and cannot be a variable.

Resources (building blocks)

Create an Azure Resource Group.

- Azure requires all resources be assigned to a resource group.

```
resource "azurerm_resource_group" "training" {
  name     = "training-workshop"
  location = "westus"
}
```

Resources (building blocks)

TIP: You can assign random names to resources.

```
resource "random_id" "project_name" {
  byte_length = 4
}
resource "azurerm_resource_group" "training" {
  name       = "${random_id.project_name.hex}-training"
  location   = "westus"
}
```

Terraform resources

Some resource types provide a special timeouts nested block argument that allows you to customize how long certain operations are allowed to take before being considered to have failed. For example, aws_db_instance allows configurable timeouts for create, update and delete operations.

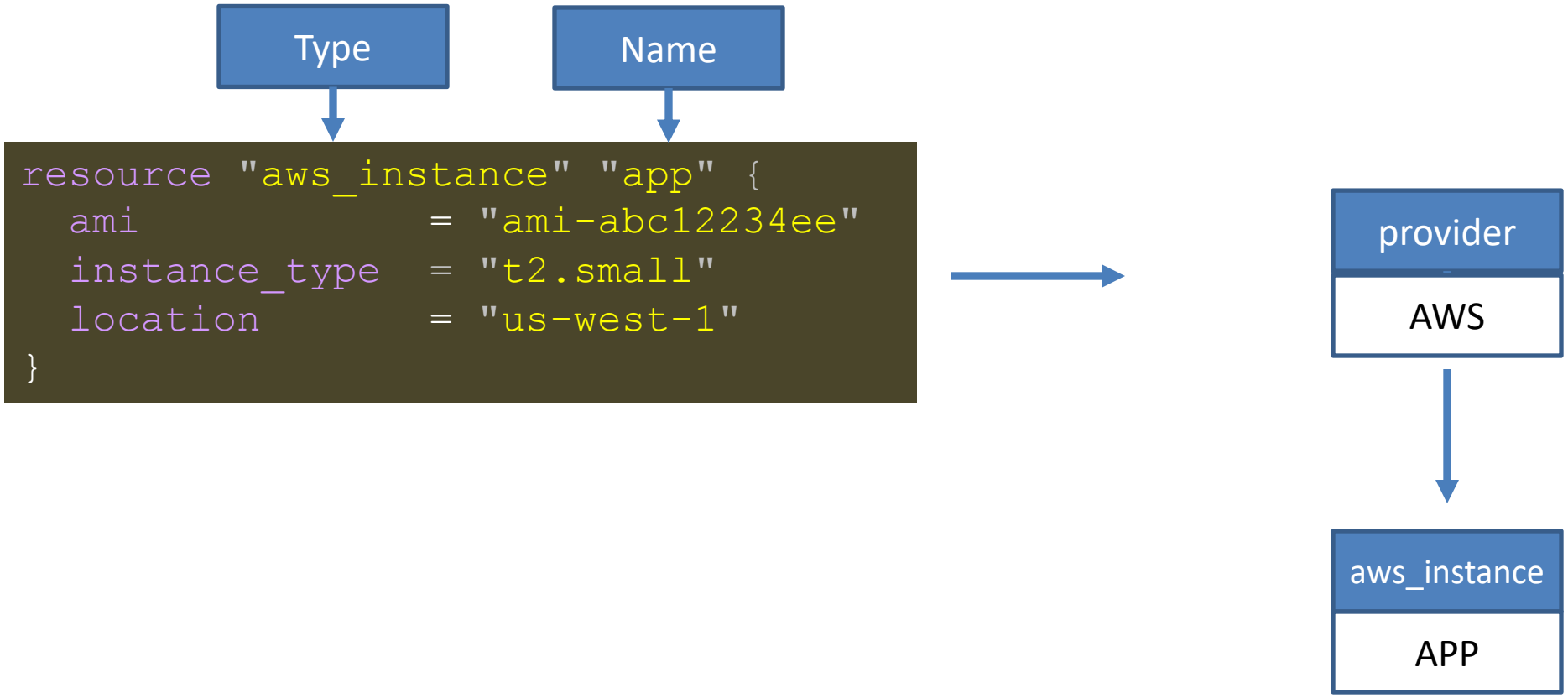
Terraform resources

```
resource "aws_db_instance" "example" {
  # ...

  timeouts {
    create = "60m"
    delete = "2h"
  }
}
```

Provider default timeouts can be overridden per operation.

Resources (building blocks)



Terraform local-only resources

While most resource types correspond to an infrastructure object type that is managed via a remote network API, there are certain specialized resource types that operate only within Terraform itself, calculating some results and saving those results in the state for future use.

For example, you can use local-only resources for:

- Generating private keys
- Issue self-signed TLS certs
- Generating random ids

The behavior of local-only resources is the same as all other resources, but their result data exists only within the Terraform state. "Destroying" such a resource means only to remove it from the state, discarding its data.

Local-Only Resources - Random

```
resource "random_id" "bucket_suffix" {
  byte_length = 8
}


resource "random_password" "db_password" {
  length = 16
  special = true
  override_special = "!#$%&*()-_=+[]{}<>:?"
}
```

Random resources generate and maintain consistent random values
across Terraform runs.

- Random strings
- Random integers
- UUIDs
- Passwords

Local-Only Resources - Time

```
resource "time_rotating" "key_rotation" {
  rotation_days = 30
}


resource "aws_kms_key" "example" {
  description = "crypto-key"
  rotation_enabled = true
}
```

Time resources help manage time-based operations and dependencies.

- Rotation triggers
- Delayed operations
- Time-based dependencies
- Schedule tracking

Terraform variables

Set default values for variables in `variables.tf`
If default values are omitted and not set anywhere else, the user will be prompted to provide them.

```
##AWS Specific Vars
variable "aws_master_count" {
  default = 10
}
variable "aws_worker_count" {
  default = 20
}
variable "aws_key_name" {
  default = "k8s"
}
```

Terraform variables

Optional variable arguments:

- default: A default value which makes the variable optional
- type: Specifies value type accepted for the variable.
- description: Specifies the variable's documentation
- validation: A block to define validation rules
- sensitive: Does not show value in output

Terraform variables (type constraint)

Type constraints are optional but highly encouraged. They can serve as reminders for users of the module and help Terraform return helpful errors if the wrong type is used.

The type argument allows you to restrict acceptable value types. If no type constraint is defined, then a value of any type is accepted.

The keyword any may be used to indicate that any type is acceptable.

If both the type and default arguments are specified, the given default value must be convertible to the specified type.

Terraform variables (type constraint)

Type constraints are created from a mixture of type keywords and constructors.

Supported type keywords:
- string
- number
- bool

Type constructors allow you to specify complex types such as collections:
- list (<TYPE>)
- set (<TYPE>
- map (<TYPE>)
- object ({<ATTR NAME> = <TYPE>, ...})
- tuple ([<TYPE>, ...])

Terraform variables

Each input variable accepted by a module must be declared using a variable block.

The label after the variable keyword is a name for the variable, which must be unique among all variables in the same module. This name is used to assign a value to the variable from outside and to reference the variable's value from within the module.

```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

Terraform variables

Terraform has some reserved variables:

- source
- version
- providers
- count
- for_each
- lifecycle
- depends_on
- locals

- These are reserved for meta-arguments in module blocks.

```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

Environment-Specific Variable Files

```
# dev.tfvars
environment = "dev"
project_id = "my-project-dev"
node_count = 1
instance_type = "t3.medium"


# prod.tfvars
environment = "prod"
project_id = "my-project-prod"
node_count = 3
instance_type = "t3.large"
```

Using separate tfvars files allows managing multiple environments with the same code base.

- One configuration, multiple environments
- Clear separation of config from code
- Easy environment promotion
- Consistent resource naming

Local Value Patterns

```
locals {
  resource_prefix = "${var.environment}-
${var.project_id}"
  common_tags = {
    Environment = var.environment
    Project = var.project_id
    ManagedBy = "terraform"
  }
  instance_name = "${local.resource_prefix}-
instance"
  bucket_name = "${local.resource_prefix}-
storage"
}
```

Local values help create consistent naming and tagging conventions across resources.

- Combine multiple variables
- Create reusable patterns
- Enforce naming conventions
- Reduce repetition

Output Management

```
output "environment_info" {
  value = {
    name = var.environment


    resources = {
      instances = aws_instance.web[*].tags["Name"]
      bucket = aws_s3_bucket.data.bucket
      vpc = aws_vpc.main.tags["Name"]
  }

  endpoints = {
    api = "https://${local.resource_prefix}-
api.example.com"
    web = "https://${local.resource_prefix}-
web.example.com"
    }
  }
}
```

Outputs expose environment-specific values while maintaining consistency across environments.

- Environment-specific values
- Consistent output structure
- Cross-environment references
- Pipeline integration points

Environment-Specific Conditions

```
locals {
  is_production = var.environment == "prod"
  instance_count = {
    dev = 1
    staging = 2
    prod = 3
  }
}
```

Use variables to create environment-specific resource configurations.

- Conditional resource creation
- Environment-based sizing
- Feature flags
- Resource counts

Terraform provisioners

Terraform is designed to programmatically create and manage infrastructure. It is not intended to replace Ansible, Chef or any other configuration management tool.

It does include provisioners as a way to prepare the system for your services.

Generic provisioners:

- file
- local-exec
- remote-exec

Provisioner Overview

```
resource "google_compute_instance" "web" {
  name = "web-server"
  machine_type = "e2-micro"


provisioner "remote-exec" {
  inline = [
    "apt-get update",
    "apt-get install -y nginx"
    ]
  }
}
```

Provisioners allow you to execute actions on local or remote machines as part of resource creation or destruction.

- Post-creation configuration
- Software installation
- File transfers
- Command execution

Terraform provisioners

Generic Provisioners:

- file
  - The file provisioner is used to copy files or directories from the machine executing Terraform to the newly created resource. The file provisioner supports both SSH and WinRM type connections.
- local-exec
  - The local-exec provisioner invokes a local executable after a resource is created. This invokes a process on the machine running Terraform, not on the resource.
- remote-exec
  - The remote-exec provisioner invokes a script on a remote resource after it is created. This can be used to run a configuration management tool, bootstrap into a cluster, etc. The remote-exec provisioner supports both SSH and WinRM type connections.

Remote-Exec Provisioner

```
resource "google_compute_instance" "web" {
  name = "web-server"

  connection {
    type = "ssh"
    user = "admin"
    private_key =
file("${path.module}/ssh/private_key")
  }

  provisioner "remote-exec" {
    inline = [
      "sudo apt-get update",
      "sudo apt-get install -y nginx",
      "sudo systemctl start nginx"
    ]
  }
}
```

Remote-exec provisioners run commands on the remote resource after creation.

- Install software packages
- Configure services
- Start applications
- Run setup scripts

Local-Exec Provisioner

```
resource "google_compute_instance" "db" {
  name = "database"


  provisioner "local-exec" {
    command = <<-EOT
    echo
"DB_HOST=${self.network_interface[0].network_ip}" >>
.env
    ./scripts/update-dns.sh ${self.name}
${self.network_interface[0].access_config[0].nat_ip}
    EOT
  }
}
```

Local-exec provisioners run commands on the machine executing Terraform.

- Generate configuration files
- Run local scripts
- Trigger external tools
- Update documentation

File Provisioner

```
resource "google_compute_instance" "app" {
  name = "application"


  connection {
    type = "ssh"
    user = "admin"
    private_key = file(var.ssh_private_key)
  }

  provisioner "file" {
    source = "configs/"
    destination = "/etc/application/"
  }
}
```

File provisioners copy files or directories to the remote resource.

- Configuration files
- Scripts
- Application code
- SSL certificates

Lab: Working with variables

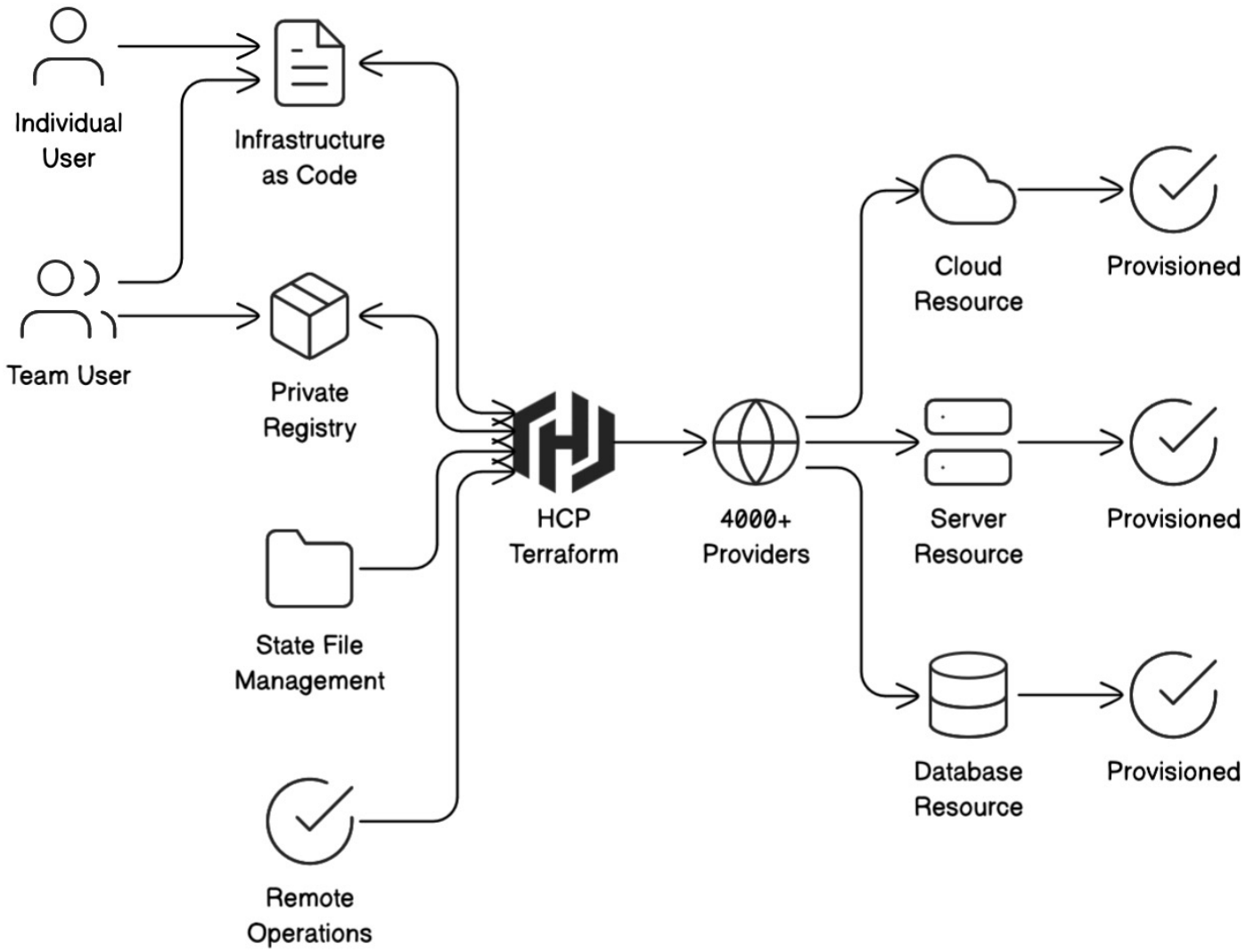
Lab: Multi resource deployment

What is HCP Terraform?

HCP Terraform is a managed service that brings the power of Terraform's infrastructure as code workflow to the cloud. Instead of running Terraform locally, HCP Terraform provides a secure, consistent, and collaborative environment for managing infrastructure. It centralizes state management, secrets, and policy enforcement, and integrates with version control systems to streamline infrastructure development.

By using HCP Terraform, teams can automate, govern, and audit their infrastructure changes with greater confidence and efficiency.

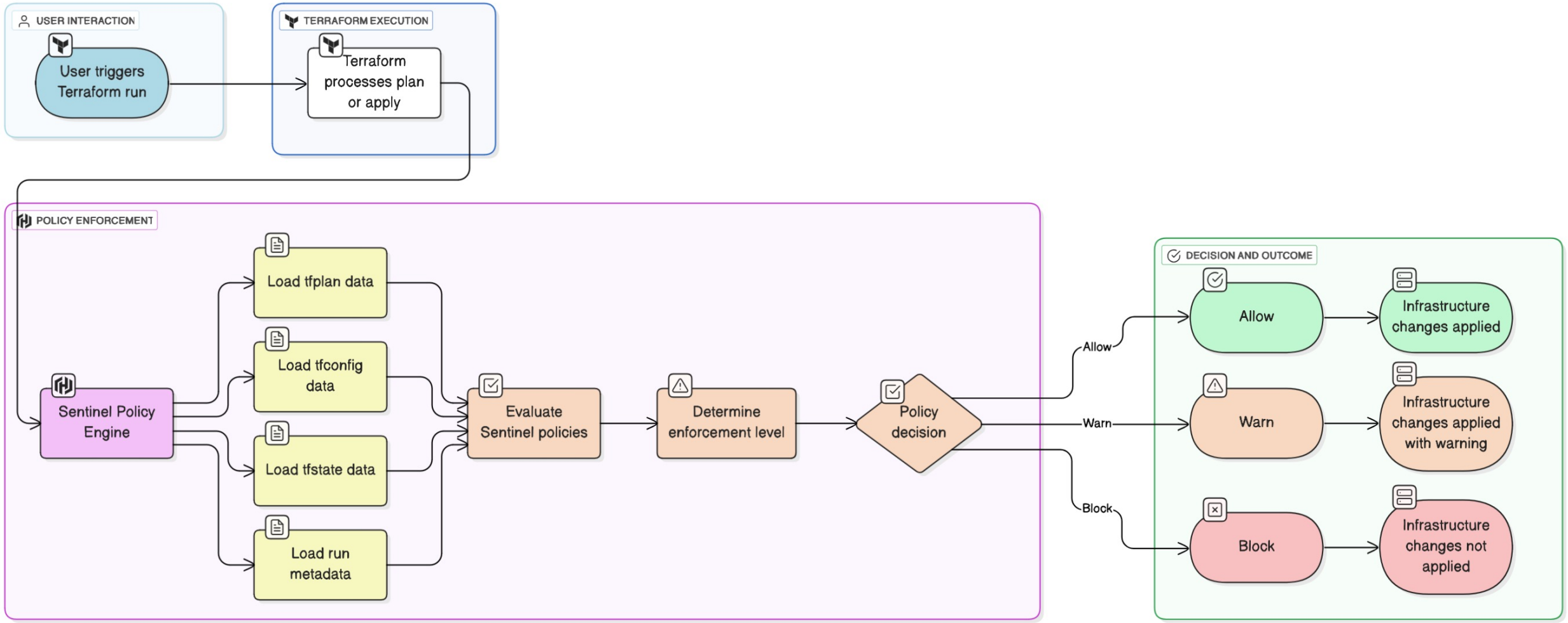
HCP Terraform Overview



Key Features of HCP Terraform

- Centralized and secure storage of Terraform state files and secrets
- Integration with version control systems (VCS) for automated runs
- Private registry for sharing modules and providers within your organization
- Enhanced visibility into infrastructure changes and operations
- Access controls, policy enforcement, and cost estimation (paid features)
- Collaboration tools for team-based infrastructure management

HCP Terraform Workflow



HCP Terraform Workflows

CLI-driven workflow:

- Run Terraform commands from your terminal, with operations executed remotely in HCP Terraform's secure environment.

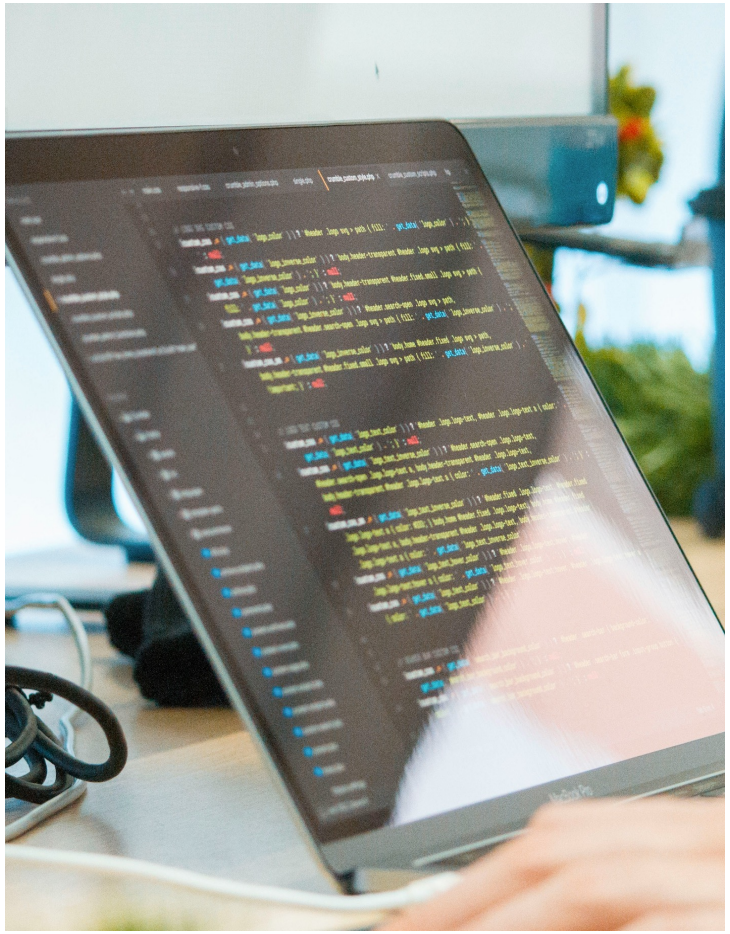
VCS-driven workflow:

- Connect a GitHub, GitLab, or Bitbucket repository; changes to code trigger runs automatically.

API-driven workflow:

- Programmatically interact with HCP Terraform using its API for custom automation and integrations.

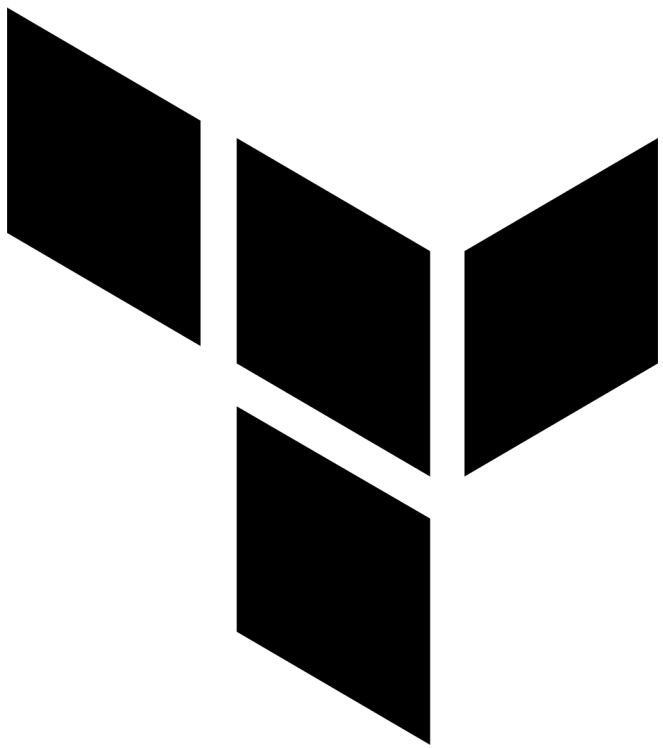
What is Terraform Enterprise?



Terraform Enterprise is HashiCorp's self-hosted distribution of HCP Terraform, designed for organizations that require full control over their infrastructure automation platform. By running Terraform Enterprise on your own infrastructure, you gain access to all the features of HCP Terraform, including advanced policy enforcement, private module registries, and detailed audit logging, while maintaining data residency and compliance with internal security requirements.

Terraform Enterprise is ideal for enterprises with strict regulatory, security, or operational needs that cannot be met by a fully managed cloud service.

Key Features of Terraform Enterprise



- Self-hosted, private instance of the HCP Terraform application
- No resource limits—scale to meet enterprise needs
- Enterprise-grade features:
- Audit logging for compliance and traceability
- SAML single sign-on (SSO) for secure user authentication
- Private module and provider registries
- Advanced access controls and team management
- Supports integration with existing enterprise systems and workflows
- Provides the same core functionality as HCP Terraform, with additional control and flexibility for on-premises or private cloud deployments

Focus of This Course – HCP Terraform

While Terraform Enterprise and HCP Terraform share many features and capabilities, this course will focus specifically on HCP Terraform, HashiCorp's managed cloud offering. HCP Terraform provides a streamlined, fully managed environment for running Terraform workflows, collaborating with teams, and enforcing policy as code—all without the operational overhead of managing your own infrastructure. Throughout this course, you'll learn how to leverage HCP Terraform's cloud-native features for secure, scalable, and collaborative infrastructure automation.

HCP Terraform CLI Workflow

- HCP Terraform supports CLI-driven, VCS-driven, and API-driven workflows.
- The CLI workflow lets you run Terraform commands locally while using HCP Terraform for secure remote execution and state management.
- To begin, authenticate your CLI by running terraform login.
- This command links your local environment to your HCP Terraform account, enabling remote plans, applies, and collaboration from the terminal.

HCP Terraform CLI Workflow

```
$ terraform login
Terraform will request an API token for app.terraform.io using your browser.

If login is successful, Terraform will store the token in plain text in
the following file for use by subsequent commands:
    /Users/redacted/.terraform.d/credentials.tfrc.json

Do you want to proceed?
  Only 'yes' will be accepted to confirm.

  Enter a value: yes
```

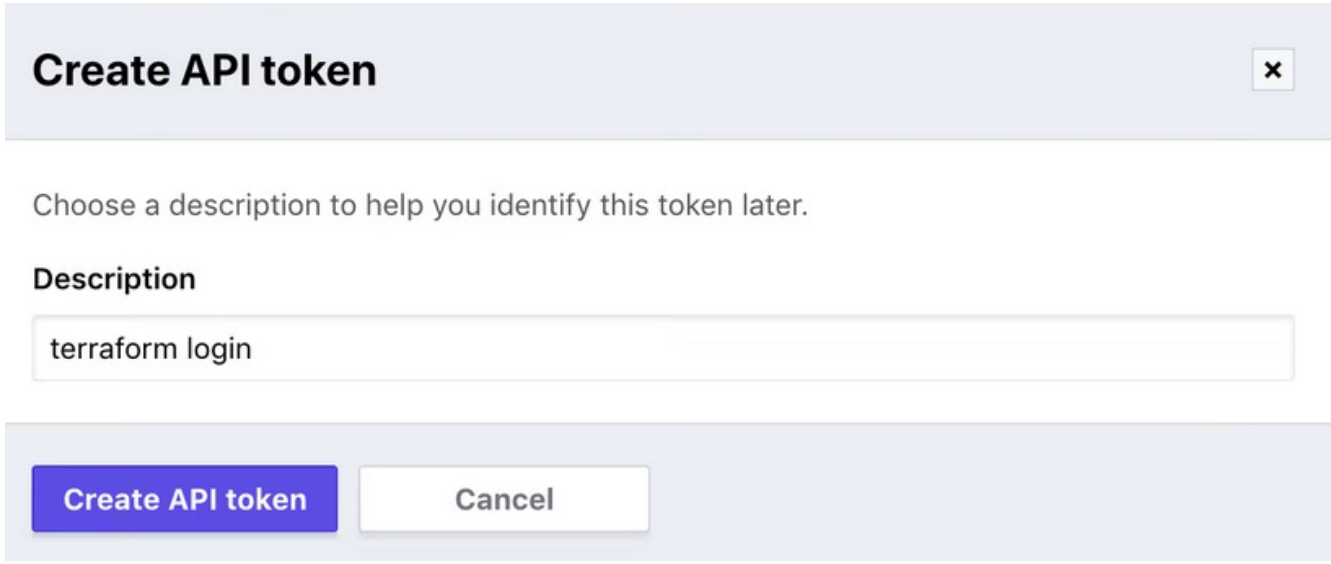
Generating and Storing your API Token

• During the CLI login process, HCP Terraform prompts you to create a new API token in the web interface.

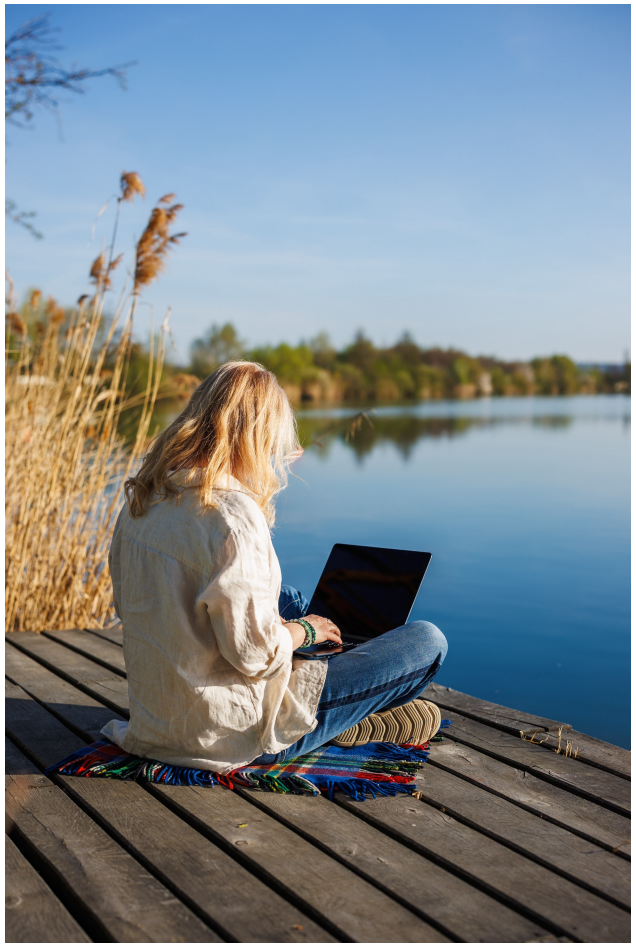
• You can give the token a descriptive name or use the default.

• Once generated, copy the token and paste it into your terminal when prompted by `terraform login`.

• The CLI stores this token in a local credentials file for future authentication.



Ready for Remote Operations



After successful authentication, your Terraform CLI is now connected to HCP Terraform. You can perform remote operations such as planning, applying, and managing state files, all within the secure and collaborative environment provided by HCP Terraform.

This setup allows you to leverage the benefits of cloud-based state management, team collaboration, and policy enforcement, while still using the familiar Terraform CLI workflow. With authentication complete, you are ready to create workspaces, manage variables, and provision infrastructure using HCP Terraform's robust cloud platform.

Lab: Authenticating to HCP Terraform