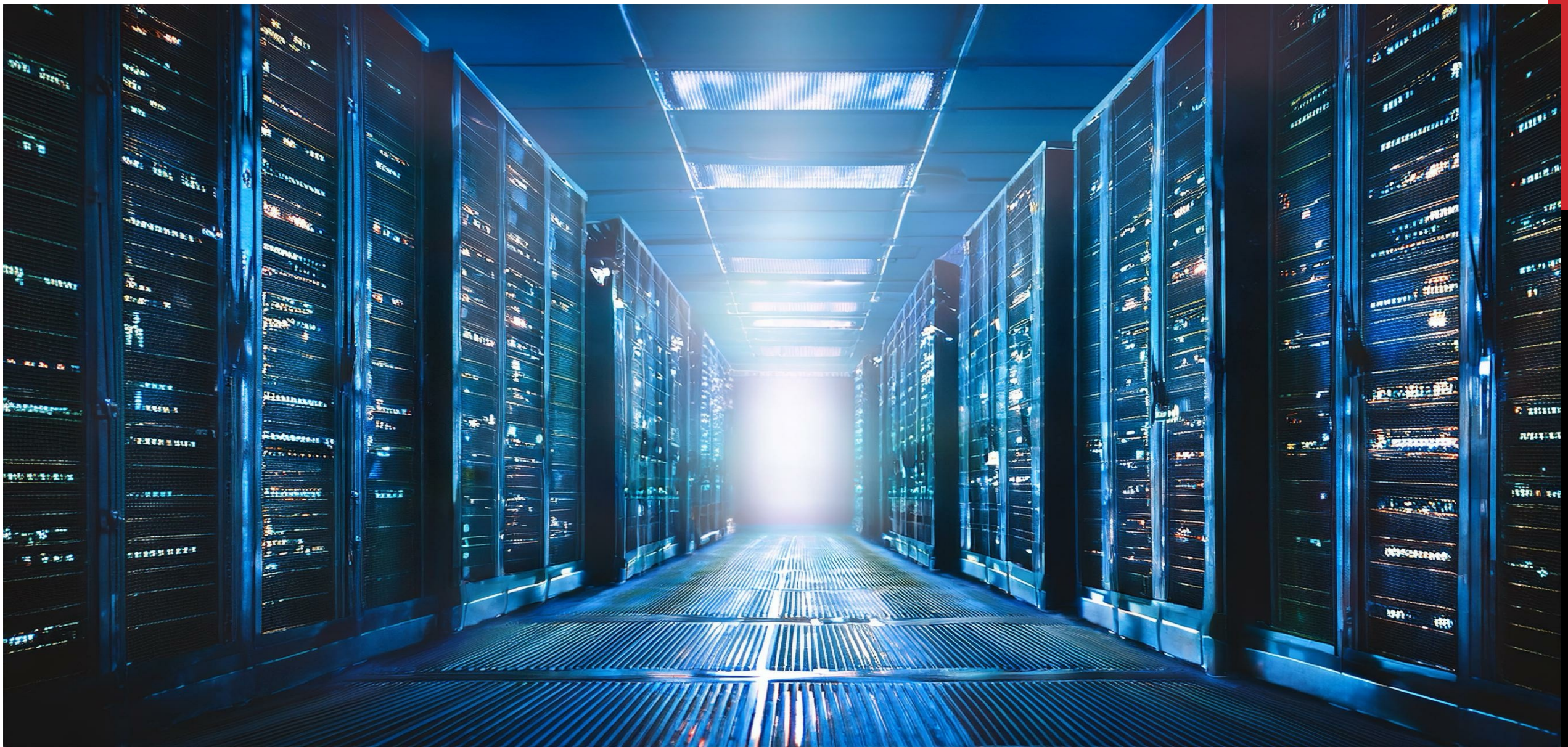
Infrastructure-as-Code & Terraform



TEKsystems Global Services

WORKFORCE DEVELOPMENT

Configuration Management

Configuration Drift

POTHOLE

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

- Manage with Infrastructure as Code (IAC)

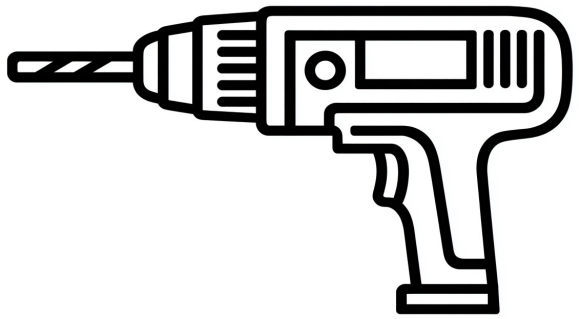
Manual

Common manual Infrastructure tasks

- Create a VPC with public and private subnets
- Manage and update IAM policies
- Provision a managed database (for example, Amazon RDS)
- Deploy an Application Load Balancer and output its DNS name

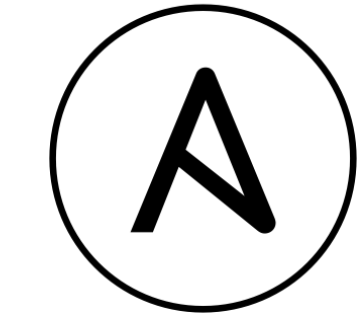
Automated

Benefits of Infrastructure Automation



- Consistent, reproducible environments
- Full Git tracking and collaboration
- Predictable changes through state
- Reusable patterns with variables and modules
- Automated cleanup and lifecycle control

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

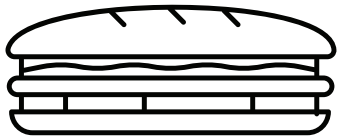
Infrastructure as Code

Provisioning infrastructure through software to achieve consistent and predictable environments.

Core Concepts

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

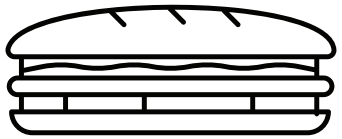
Imperative



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"
  ]
}
```

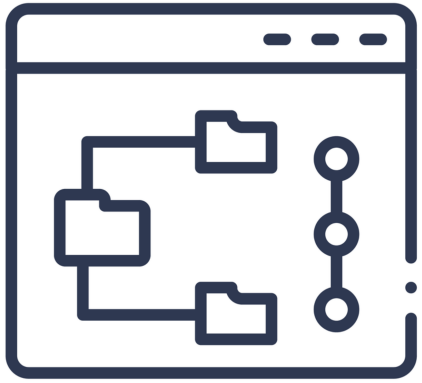
POP QUIZ:

What challenges does Infrastructure as Code solve?

Terraform Introduction

Automating Infrastructure

Provisioning resources

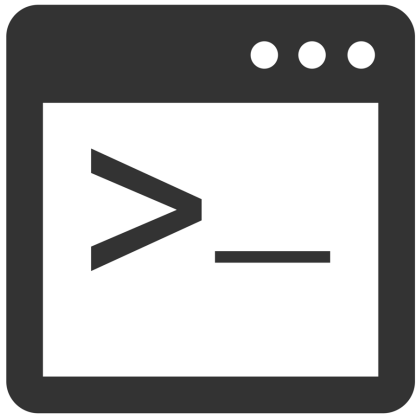


Version Control

Plan Updates

Reusable Templates

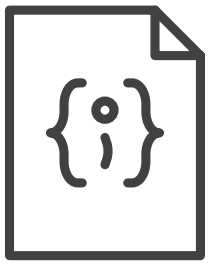
Terraform components



Terraform executable

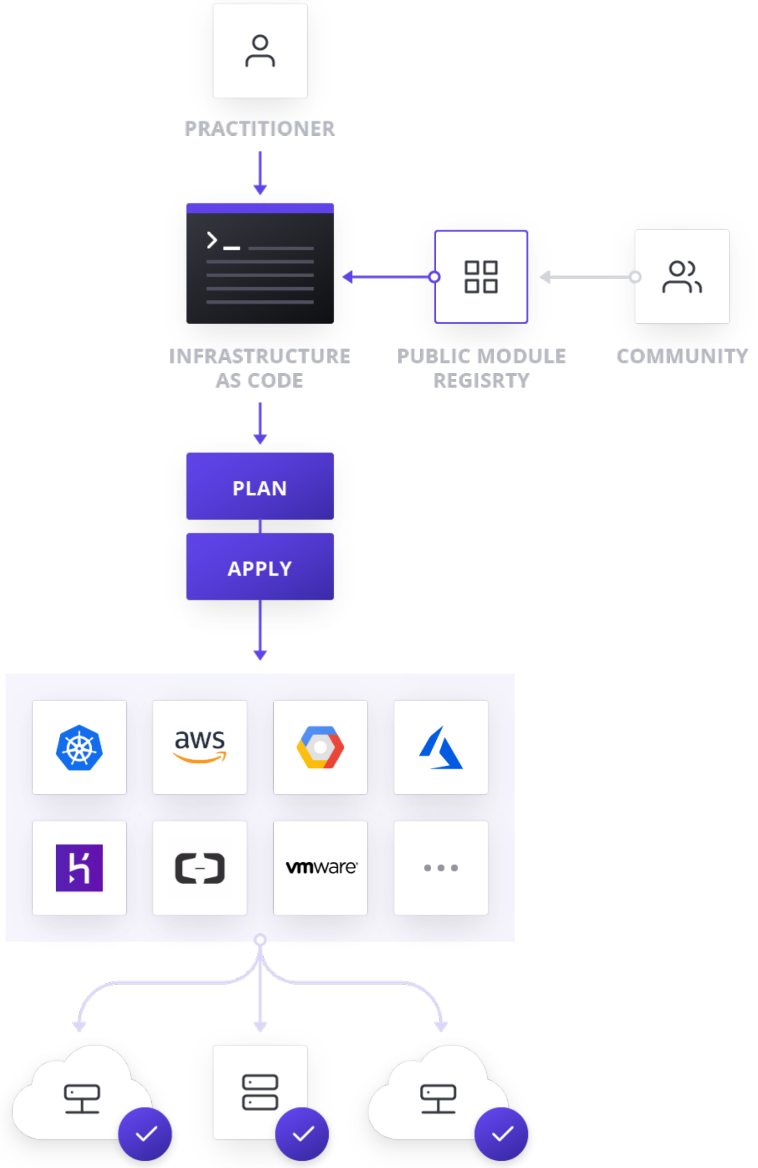
Terraform files

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: Terraform Setup

Lab: Terraform First Instance

Terraform configuration

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

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

Terraform configuration

```
data "aws_ami" "ubuntu" {
  most_recent = true
  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-*"]
  }
  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
  owners = ["099720109477"] # Canonical
}
```

- Data sources
  - Queries AWS API for latest Ubuntu 16.04 image.
  - Stores results in a variable which is used later.

Terraform configuration

```
resource "aws_instance" "aws-k8s-master" {
  subnet_id              = "${var.aws_subnet_id}"
  depends_on             = ["aws_security_group.k8s_sg"]
  ami                    = "${data.aws_ami.ubuntu.id}"
  instance_type          = "${var.aws_instance_size}"
  vpc_security_group_ids = ["${aws_security_group.k8s_sg.id}"]
  key_name               = "${var.aws_key_name}"
  count                  = "${var.aws_master_count}"
  root_block_device {
    volume_type           = "gp2"
    volume_size           = 20
    delete_on_termination = true
  }
  tags {
    Name = "k8s-master-${count.index}"
    role = "k8s-master"
  }
}
```

- Resource: Defines specifications for creation of infrastructure.

Terraform configuration

```
resource "aws_instance" "aws-k8s-master" {
  subnet_id              = "${var.aws_subnet_id}"
  depends_on             = ["aws_security_group.k8s_sg"]
  ami                    = "${data.aws_ami.ubuntu.id}"
  instance_type          = "${var.aws_instance_size}"
  vpc_security_group_ids = ["${aws_security_group.k8s_sg.id}"]
  key_name               = "${var.aws_key_name}"
  count                  = "${var.aws_master_count}"
  root_block_device {
    volume_type           = "gp2"
    volume_size           = 20
    delete_on_termination = true
  }
  tags {
    Name = "k8s-master-${count.index}"
    role = "k8s-master"
  }
}
```

- ami is set by results of previous data source query

Terraform configuration

```
##AWS Specific Vars
variable "aws_master_count" {
  default = 10
}
variable "aws_worker_count" {
  default = 20
}
variable "aws_key_name" {
  default = "k8s"
}
variable "aws_instance_size" {
  default = "t2.small"
}
variable "aws_region" {
  default = "us-west-1"
}
```

- Define sane defaults in variables.tf

Terraform configuration

```
resource "aws_instance" "aws-k8s-master" {
  subnet_id              = "${var.aws_subnet_id}"
  depends_on             = ["aws_security_group.k8s_sg"]
  ami                    = "${data.aws_ami.ubuntu.id}"
  instance_type          = "${var.aws_instance_size}"
  vpc_security_group_ids = ["${aws_security_group.k8s_sg.id}"]
  key_name               = "${var.aws_key_name}"
  count                  = "${var.aws_master_count}"
  root_block_device {
    volume_type           = "gp2"
    volume_size           = 20
    delete_on_termination = true
  }
  tags {
    Name = "k8s-master-${count.index}"
    role = "k8s-master"
  }
}
```

- Value is 'count' is setup by variable in variables.tf

Terraform CLI

The `terraform plan -refresh-only` command is used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure. This can be used to detect any drift from the last-known state, and to update the state file.

This does not modify infrastructure but does modify the state file. If the state is changed, this may cause changes to occur during the next `plan` or `apply`.

Terraform CLI

Terraform Apply with -var

The -var option lets you override variable values at runtime without editing your .tf files. It is useful for quick tests, one-off deployments, and values you do not want hardcoded into your Terraform configuration.

- Allows you to supply variable values directly from the command line
- Can override defaults in variables.tf without changing any code

```
terraform apply \
  -var="environment=prod" \
  -var="instance_type=m5.large"
```

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.

HashiCorp Configuration Language (HCL)

ARM JSON:

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

Terraform:

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

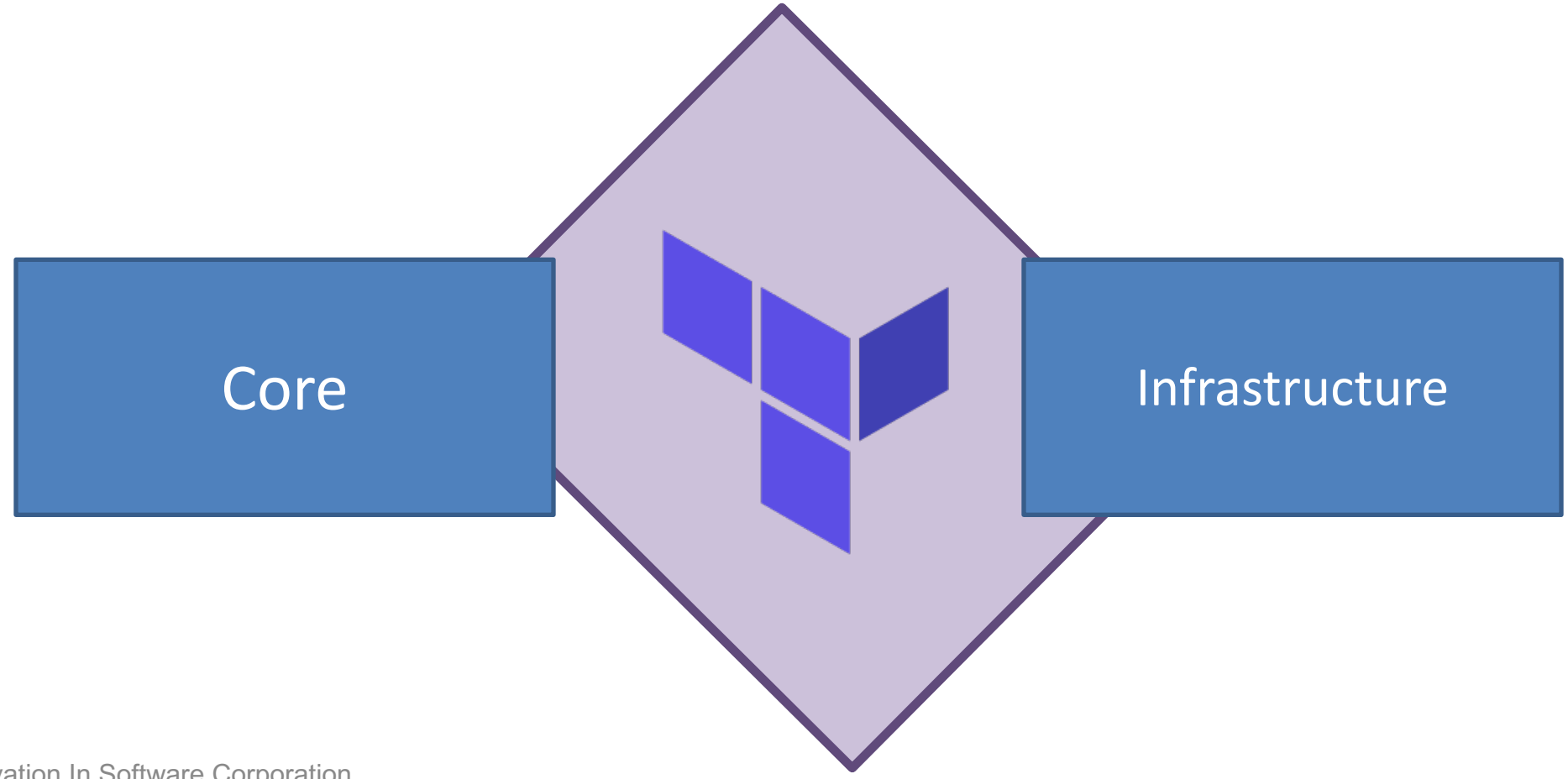
Terraform code (HCL) is "easy" to learn and "easy" to read. It is also more compact than equivalent JSON configuration.

Terraform model

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

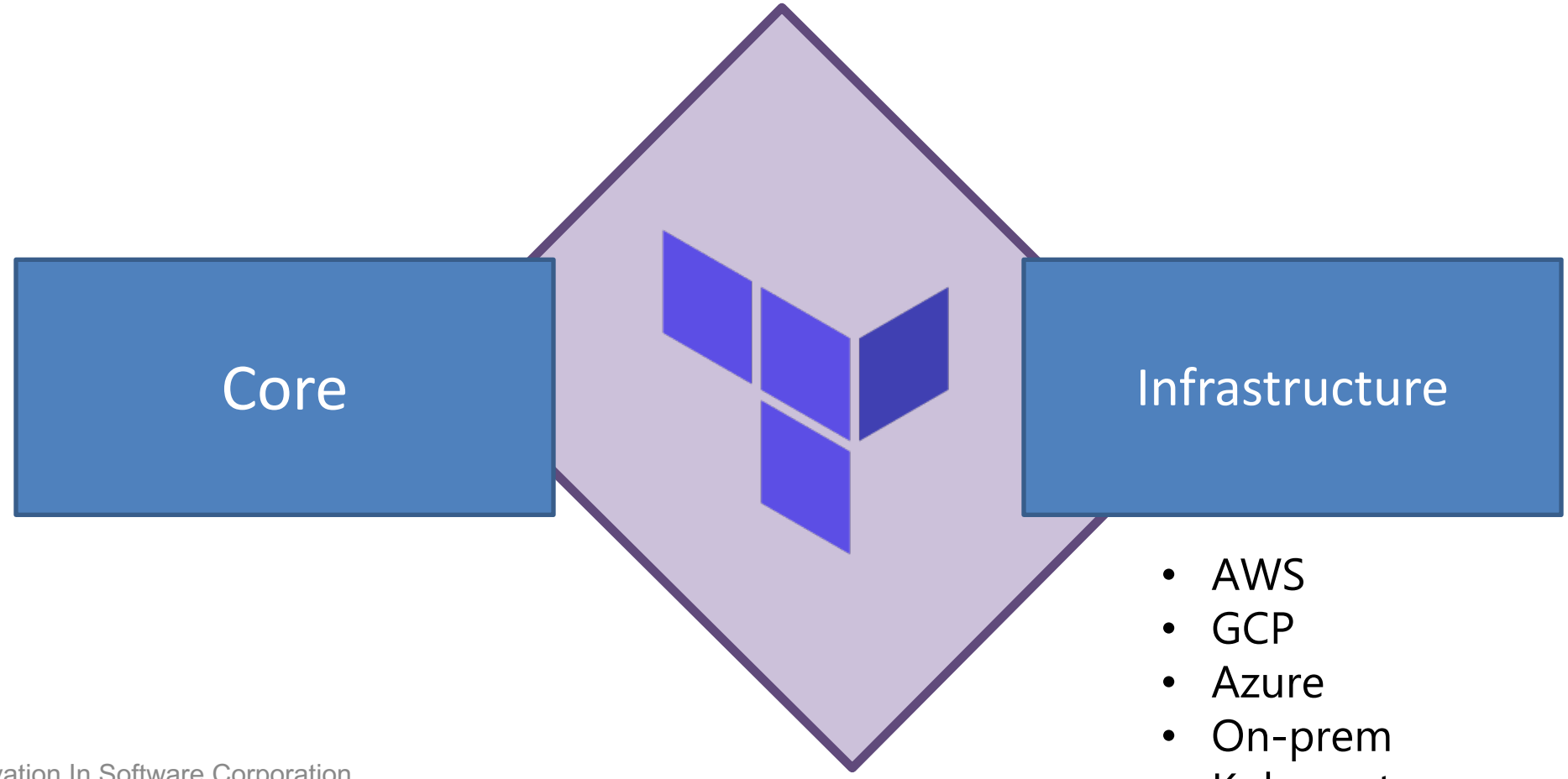
Terraform model

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

Terraform Resources

Resource Types Overview

```
# Cloud provider resource
resource "aws_instance" "web" {}


# Meta-resource
resource "terraform_remote_state" "network" {}


# Local-only resource
resource "null_resource" "example" {}
```

Resources in Terraform represent different kinds of infrastructure components, each with their own specific configuration patterns.

- Provider-specific resources (aws_instance)
- Meta-resources (terraform_remote_state)
- Null resources for local operations
- Data resources for read-only operations

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

Resource Arguments

```
resource "aws_instance" "web" {
  # Required arguments
  ami = "ami-0c02fb55956c7d316" # Amazon Linux 2 AMI
in us-east-1
  instance_type = "t2.micro"

  # Meta-arguments
  count = 3

  tags = {
    Name = "web-server"
  }
}
```

Resources accept different types of arguments that define their configuration and behavior.

- Required arguments (name, type)
- Optional arguments with defaults
- Computed arguments (set by provider)
- Meta-arguments (count, for_each)

Lab: Build first instance

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.

Terraform local-only resources

Local-only resources are also referred to as logical resources

This is primarily used for easy bootstrapping of throwaway development environments.

```
resource "tls_private_key" "example" {
  algorithm   = "ECDSA"
  ecdsa_curve = "P384"
}
```

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

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

When you declare variables in the root module of your configuration, you can set their values using CLI options and environment variables. When you declare them in child modules, the calling module should pass values in the module block.

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"
    }
  ]
}
```

Terraform Variable Data Types

Terraform supports several data types that help you control how values are structured and validated in your configuration. These types make your infrastructure code predictable, readable, and easier to maintain.

Terraform Variable Data Types

Primitive Types

- **String -** A sequence of characters. Used for names, IDs, instance types, and similar text values.
- **Number -** Any numeric value, including integers and decimals.
- **Bool -** True or false values used for enabling or disabling features.

Terraform Variable Data Types

Collection Types

- **list(T)** - An ordered list of values, all of the same type. Common for CIDR lists or subnet lists.
- **set(T)** - A collection of unique values. Order does not matter.
- **map(T)** - A group of key value pairs, similar to dictionaries.

Terraform Variable Data Types

Structural Types

- **object({})** - A custom structure with named attributes. Helpful for grouping related fields.
- **tuple([ ]) -** A fixed sequence of values where each element can have a different type.
- **Dynamic Type (any)** - Accepts any type and gives full flexibility when structure is unknown.

Lab: variables

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

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

Lab: Working with variables

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

Output

```
output "instance_id" {
  description   = "ID of the EC2 instance"
  value         = aws_instance.lab2-tf-example.id
}


output "instance_private_ip" {
  Description = "Private IP of EC2 instance"
  value = aws_instance.lab2-tf-example.private_ip
}
```

Outputs give you a clean way to display important information after a Terraform run. They help you quickly see values your infrastructure created.

- Show important info after terraform apply
- Easy to reference with terraform output
- Can be marked **sensitive** if needed

Module Variables

```
module "ec2_instances" {
  source = "./modules/aws-instance"

  # setting module variables
  instance_count  = 2
  instance_type      = var.instance_type

  tags = {
    project     = "project-alpha",
    environment = "dev"
  }
}
```

Modules have their own variables. You **cannot** pass values directly into a module using `-var`.

Instead, you set the module's variable values **inside the module block**, and *those* can use parent variables (including ones provided with `-var`).

Terraform State Files

Terraform creates a few files to track and protect your infrastructure state:

- **terraform.tfstate** – The main file that stores the current state of your resources.
- **terraform.tfstate.backup** – A backup copy kept automatically in case something goes wrong.
- **Lock file (.terraform.lock.hcl)** – Records provider versions to ensure consistent runs across environments.

Lab: More Variables