HashiCorp Terraform

The Ultimate Beginner's Guide

Bryan Krausen

Why Learn Terraform?

66% of Organizations are Increasing Cloud Spend YoY

Organizations are continuing to migrate workloads to one or more public cloud providers.

79% of Organizations have or are Planning Multi-Cloud Deployments

Management of multi-cloud environments requires cloud-agnostic solutions.

64% of Organizations are Experiencing a Shortage of Skilled Staff

Don't have the staff expertise they need to support their cloud infrastructure strategy.

Source: HashiCorp State of Cloud Strategy Survey

How This Course Will Transform Your Skills

Gain Real Skills, Build Confidence, and Open New Doors in Cloud Infrastructure

Hands-On Learning, Fast Results

Get practical experience with real-world Terraform scenarios from the start. You'll build and manage infrastructure right away, gaining skills you can apply immediately.

Confidence in Infrastructure Automation

By the end, you'll feel confident using Terraform to automate and manage cloud infrastructure, laying a solid foundation for advanced cloud projects.

Unlock Opportunities in Your Cloud Career

Terraform is a critical skill in the cloud ecosystem, and this course will give you the edge to stand out, whether you're looking to advance or break into cloud roles.

☝ HASHICONF 2024

INSTRUCTOR

Bryan Krausen

Terraform AUTHORIZED INSTRUCTOR

Welcome! I'm a Premier Instructor, a HashiCorp Ambassador, and a seasoned technologist who is passionate about cloud infrastructure and security. I help organizations realize the potential of HashiCorp's tools.

btk.me/btk

Why Trust Me?

I Consult with VERY Large Organizations

During my (many) years of consulting on HashiCorp, I've designed, implemented, fixed, and managed it all ☺

I'm VERY Involved with the Community

I've spoken at 10+ HashiCorp User Groups (HUGs), 5 x HashiConf conferences, and I am one of three people who has been a HashiCorp Ambassador since the start

I Help Write HashiCorp Certification Exams

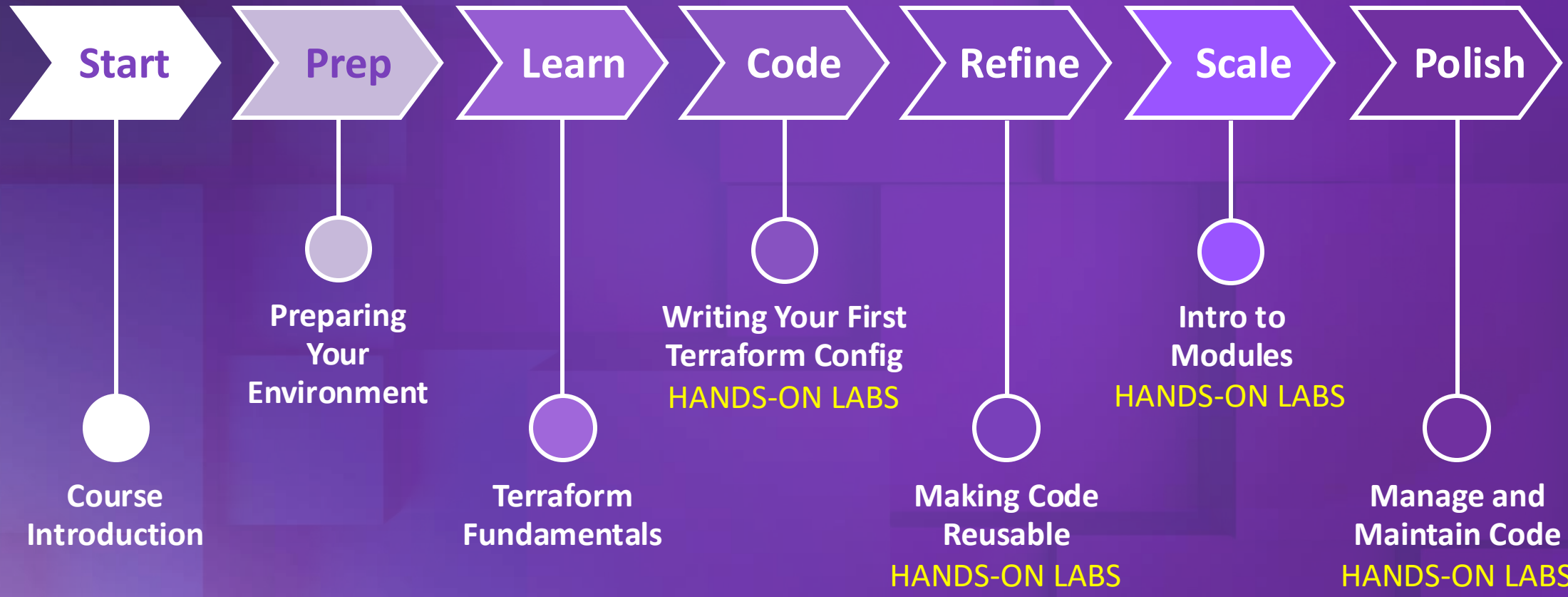
I've contributed to EVERY single certification exam that HashiCorp offers and continue to do so...

How Do We Get There?

A Variety of Content to Teach, Show, and Get you Hands-On Experience

- **Lectures to Build Understanding**
- **Demos for Real-World Examples**
- **Hands-On Labs to Apply Skills**

Course Topics and Timeline



Access All Course Code on GitHub

All code used in this course is published in a GitHub repository, so you can follow along, review, and experiment with every example provided.

github.com/btkrausen/terraform

HashiCorp Terraform
Hands-On Labs

Get access to FREE hands-on labs to practice with Vault configurations, settings, and components...all right in your browser using GitHub Codespaces.

github.com/btkrausen/terraform-codespaces

Learning Terraform: Tips for Your Journey

Start Small and Build Gradually

Learning Terraform can feel overwhelming but remember that every small step adds up.

Mistakes Are Part of the Process

Errors are expected and even helpful. Don't be discouraged if things don't work right away. Debugging is a great way to deepen your understanding.

Ask Questions and Seek Community

You're not alone on this journey. Reach out for help when you need it, whether in the Q&A, forums, or with peers.

LET'S GET! STARTED!

LECTURE

Introduction to HashiCorp Terraform

What is Terraform?

Created by HashiCorp

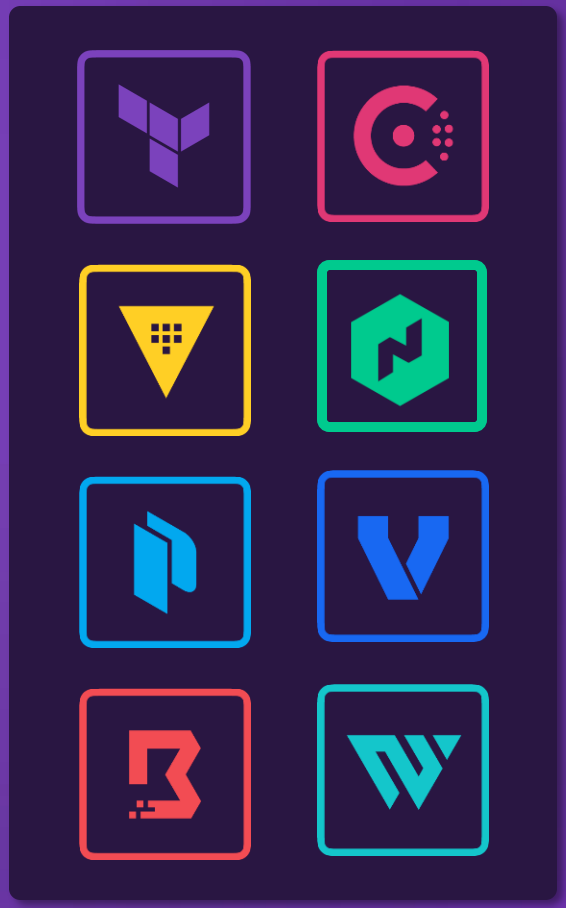
Often more famous by its tools rather than its name. It also offers many industry-standard solutions

Declarative Approach

Focuses on describing "what" needs to be done rather than explicitly stating "how" to do it, allowing the system to determine the best execution steps

Based on HashiCorp Configuration Language (HCL)

A human-readable language used in Terraform to define infrastructure resources in a clear, structured, and reusable way



What is Terraform?

The industry standard Infrastructure as Code (IaC) tool that automates the provisioning, management, and versioning of infrastructure.

Codify Your Infrastructure

Rather than clicking around in a console, create a repeatable, versioned workflow to provision and manage resources

Standardize Workflows

Use a single tool to deploy and manage resources required for application workloads. Quickly scale up using modules and other features.

Platform Agnostic

Deploy infrastructure as code using HCL to provision resources from any cloud or datacenter.

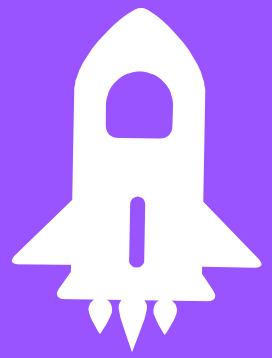
Benefits of Terraform

VERSION CONTROL AND AUDITIBILITY

Infrastructure changes are tracked in version control systems, providing a clear history and ability to revert

AUTOMATION AND EFFICIENCY

Automate the provisioning and management of infrastructure, reducing manual tasks and accelerating deployment times



SCALABILITY & FLEXIBILITY

Quickly scale infrastructure up or down to adapt to changing requirements by modifying code instead of manually changing resources.

COLLABORATION

Teams can collaborate on infrastructure changes just like application code, improving transparency and reducing silos in operations

CONSISTENCY & REPEATABILITY

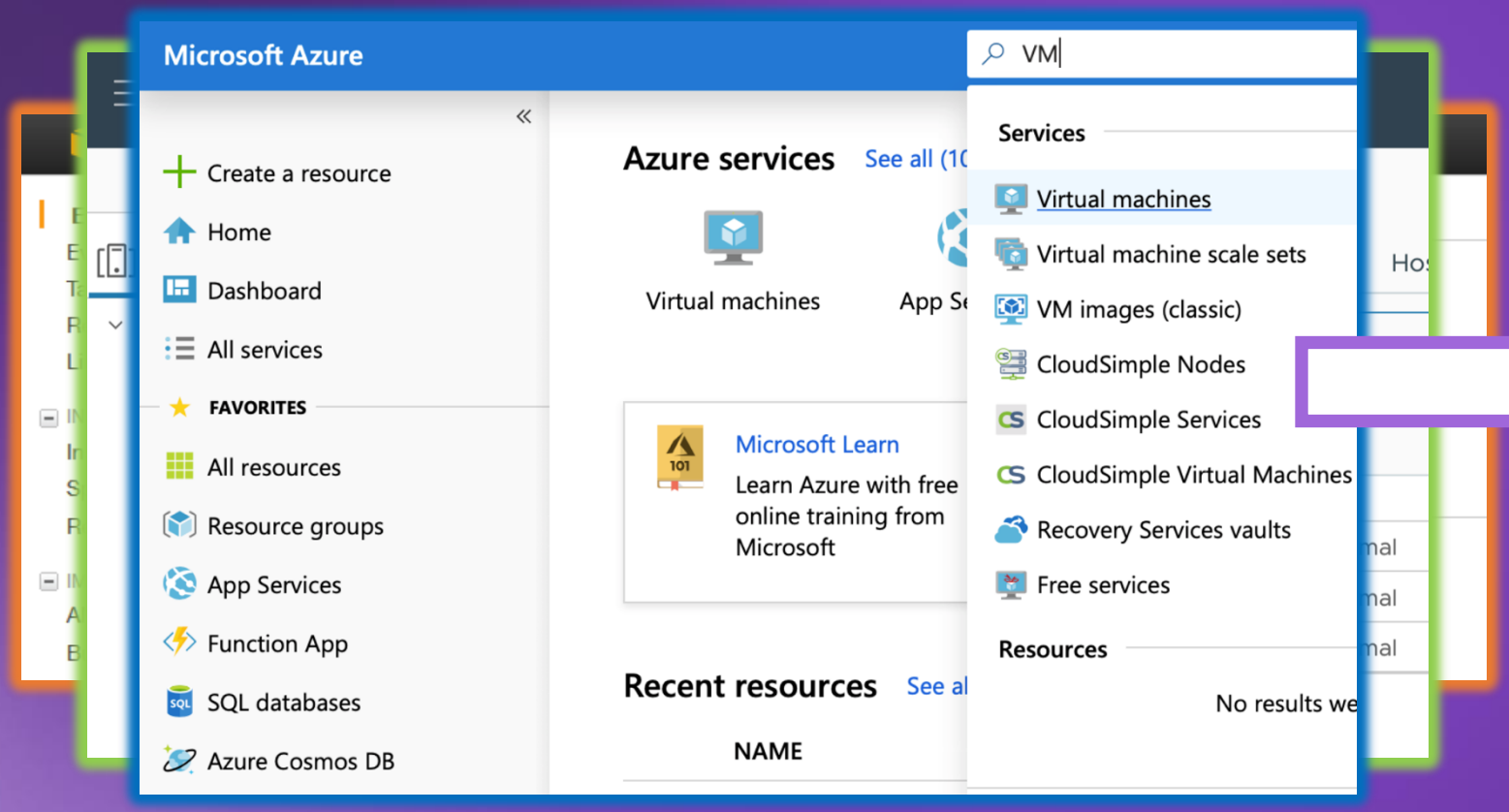
Ensures that infrastructure is consistently deployed across environments, reducing configuration drift and human errors

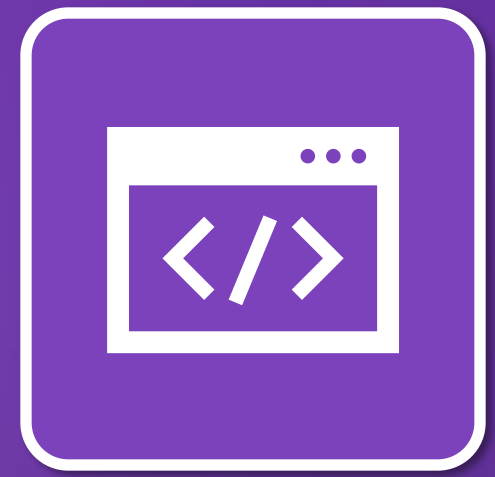
Terraform is Platform Agnostic

Before Using Terraform, You Need to First Be Proficient on the Target Platform

While it might feel like it, Terraform isn't magic. You need to understand the target platform before using Terraform.

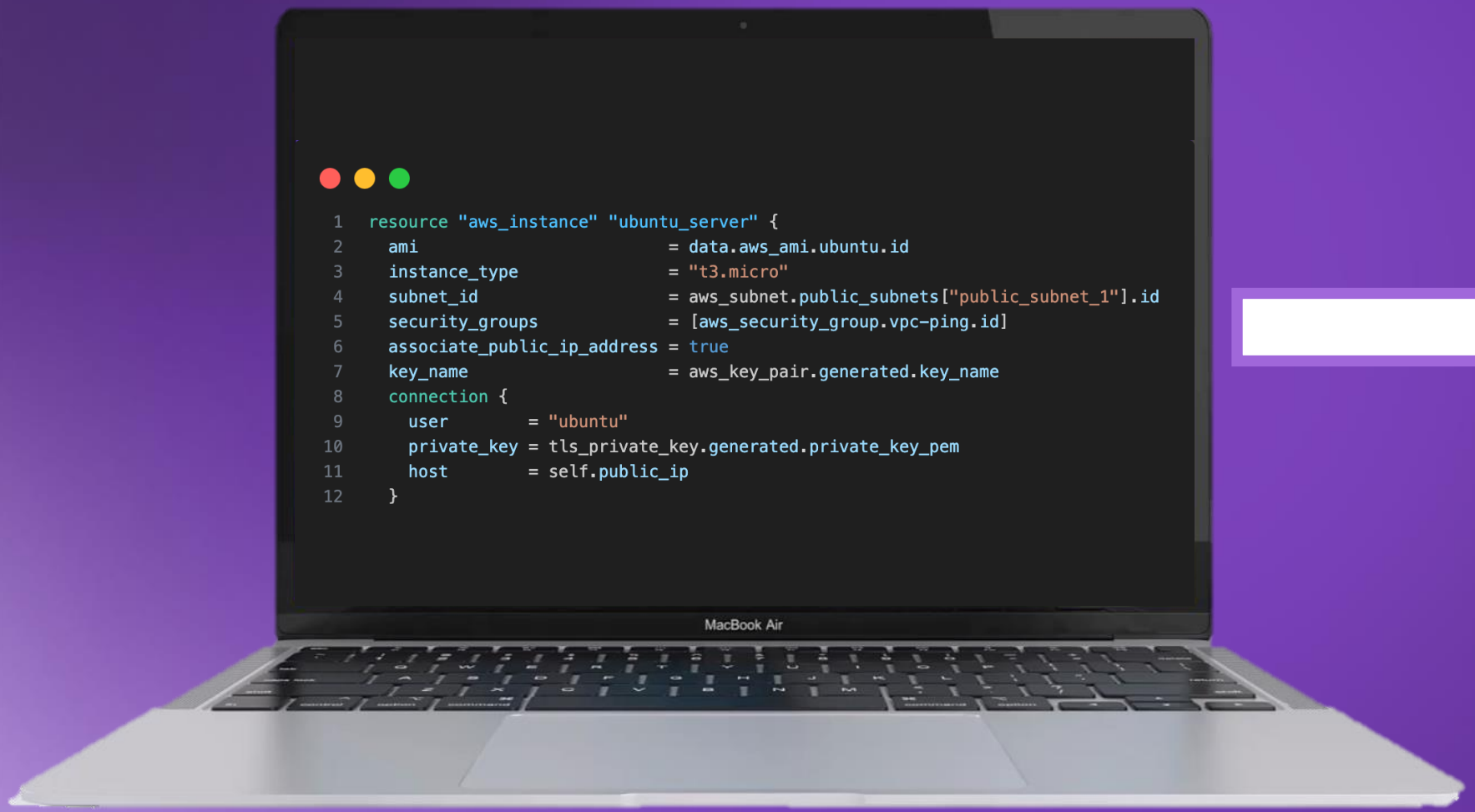
Traditional Process of Provisioning Resources





- ✓ Virtual Machine
- ✓ Container
- ✓ Network

Provisioning with Terraform



Application Workload

Reusing Code with Terraform

Input Variables

AMI = Ubuntu
Subnet = 10.16.7.0
App = Web Server

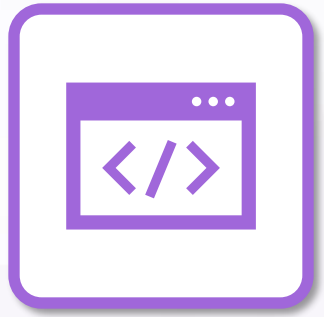
```
resource "aws_instance" "basic_server" {
  ami                         = data.aws_ami.image.id
  instance_type               = var.instance_size
  subnet_id                   = aws_subnet.public_subnets.id
  security_groups             = [aws_security_group.ingress-ssh.id]
  associate_public_ip_address = true
  key_name                    = aws_key_pair.generated.key_name
  }
```

Web Application

Input Variables

AMI = Windows
Subnet = 10.0.3.0
App = Data Processing

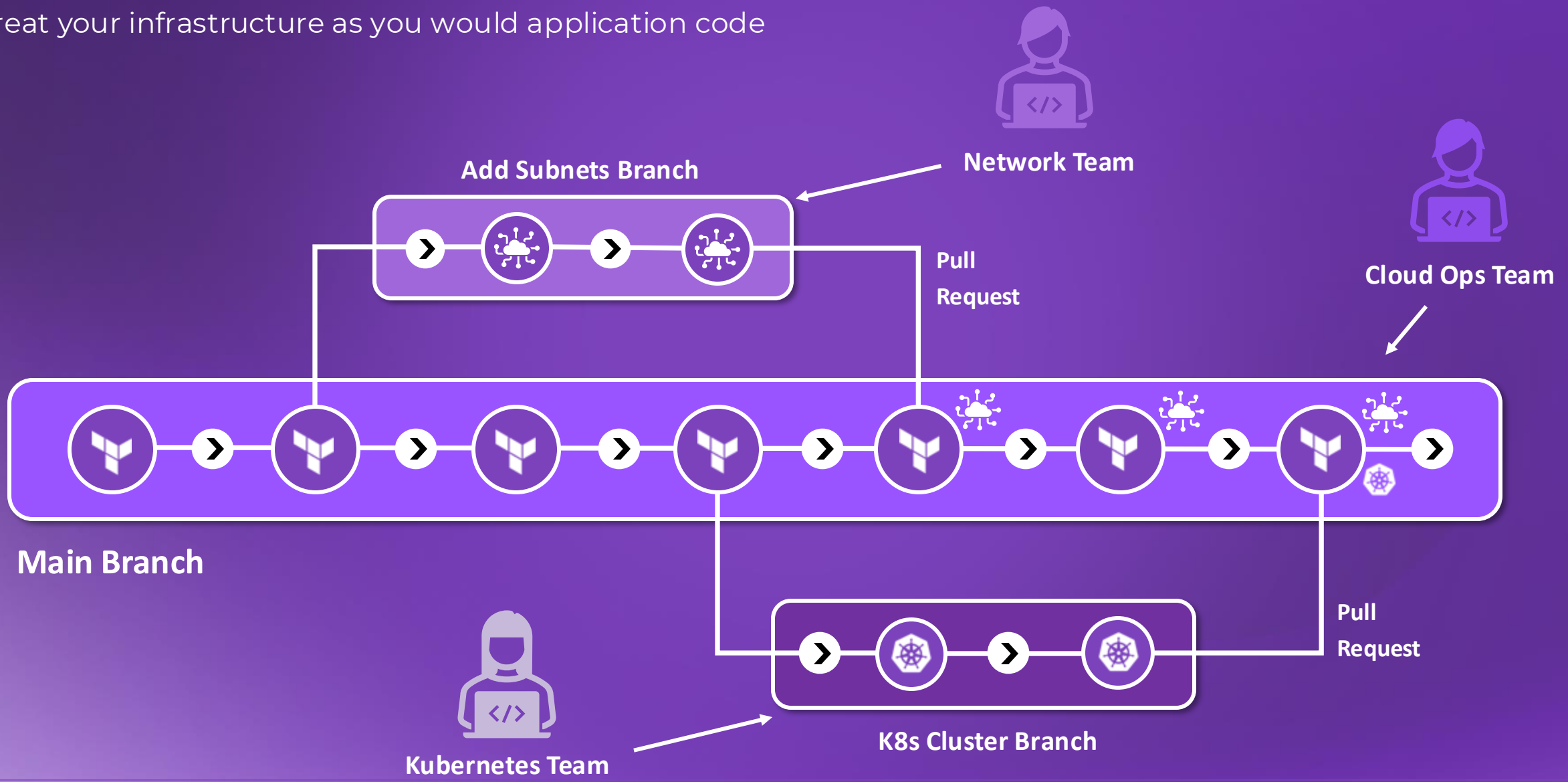
```
resource "aws_instance" "basic_server" {
  ami                         = data.aws_ami.image.id
  instance_type               = var.instance_size
  subnet_id                   = aws_subnet.public_subnets.id
  security_groups             = [aws_security_group.ingress-ssh.id]
  associate_public_ip_address = true
  key_name                    = aws_key_pair.generated.key_name
  }
```



Data Application

Infrastructure as Code

Treat your infrastructure as you would application code



Speed & Consistency

Terraform can help you quickly spin up and decommission infrastructure for development, testing, QA, and production.

By defining infrastructure as code, you can reduce the risk of mistakes. You can create standardized components that can be integrated across projects consistently.

You can save infrastructure resources and their dependencies as code in a specified environment.

Core Components

Terraform Core
CLI tool that provisions and manages infrastructure resources as defined in Terraform configuration files

Providers
Extends the functionality of Terraform for specific platforms, such as public cloud providers, SaaS offerings, etc.

Resources
Infrastructure components or services that are managed by Terraform (virtual machines, networks, DNS records, etc.)

State
How Terraform maps the desired configuration with real-world resources on the target platform

Modules
Reusable and sharable blocks of code that can be called over and over again

Terraform Editions

Terraform Community — Command-Line Based

HCP Terraform — SaaS Platform

Terraform Enterprise — Self-Hosted & Managed

Compare Terraform with Other Tools

Terraform is often confused with other popular tools commonly found in many organizations.

Let's set the record straight...

Similar to Terraform

AWS CloudFormation

Azure Bicep

Pulumi

Infrastructure as Code Tools

- Infrastructure Provisioning and Management
- Declarative Language
- Immutable Infrastructure
- Modular Design

Different but Complementary

Configuration Management Tools

- Installing Packages
- Managing Files
- Manage Configuration Systems
- Automating Routine Tasks
- Orchestrating Multi-Step Tasks

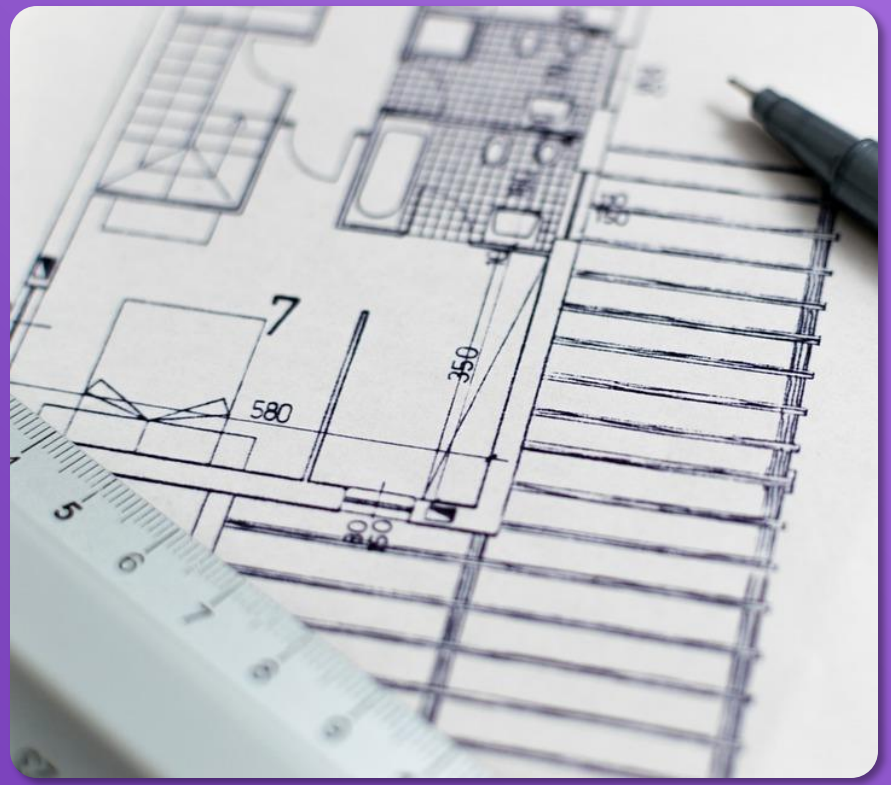
What Does It Mean to Be Declarative?

A declarative approach focuses on defining *what* you want the outcome to be, not *how* to achieve it.

DECLARATIVE vs IMPERATIVE

How Terraform Defines Desired State

The desired state is the blueprint of your infrastructure: resources, properties, and relationships.



What you write in your Terraform configuration is known as the desired state. You are telling Terraform "Create resources exactly how I've defined it within my Terraform configuration."

Terraform compares the current state of 'real-world' resources with the desired state and makes the appropriate changes.

How Do I Get Started with Terraform?

1. **Download and Install Terraform**
   Get the proper binary for your machine.

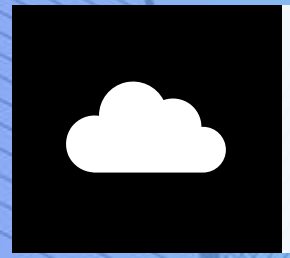
2. **Prepare Your Workspace**
   Install your favorite IDE, create directories, etc.

3. **Obtain Credentials for Target Platform**
   Terraform needs a way to communicate with your cloud

4. **Write Your First Terraform Config**
   Start with basic resources and iterate from there.

SUMMARY

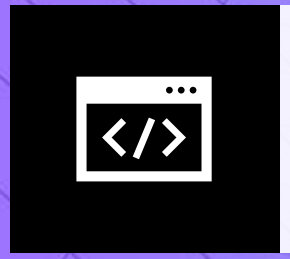
Terraform is a tool created by HashiCorp that is used to manage infrastructure as code.



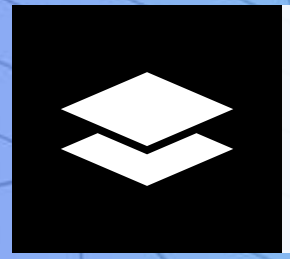
Terraform is platform-agnostic. It can be used to manage infrastructure on any cloud platform or data center.

Terraform takes a declarative approach, which means that users describe what they want and Terraform will figure out

Benefits of Terraform include version control, scalability, increased collaboration, and repeatability



Terraform uses the HashiCorp Configuration Language (HCL) to define infrastructure



Terraform's core components include Terraform Core, Providers, Resources, State, and Modules.

LECTURE

Learn the Basics of HCL

HashiCorp Configuration Language (HCL)

Terraform configuration language to describe the infrastructure that Terraform manages.

Declarative Language

Used in Terraform (and other HashiCorp products) to define infrastructure configurations

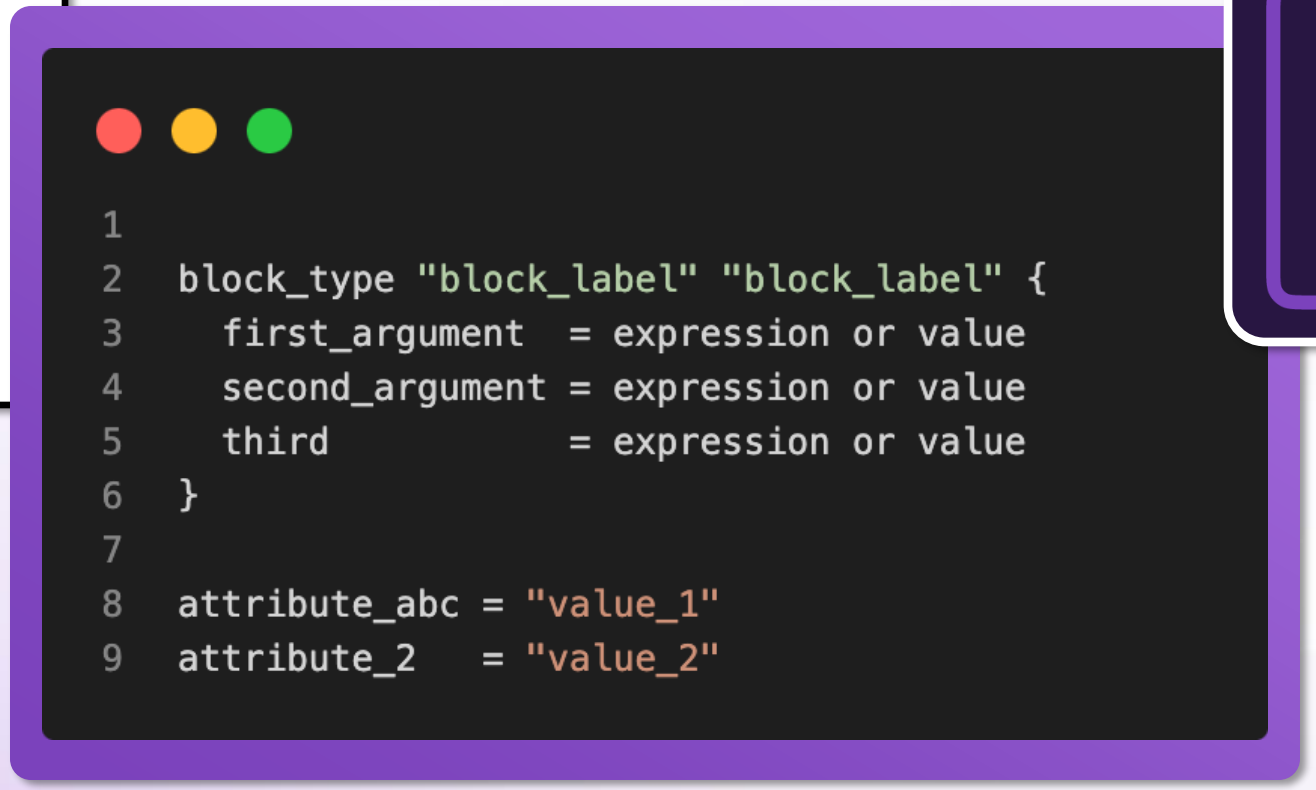
Easy to Read and Write

HCL provides a clean and human-readable syntax for expressing resources and dependencies

Terraform's Primary Interface

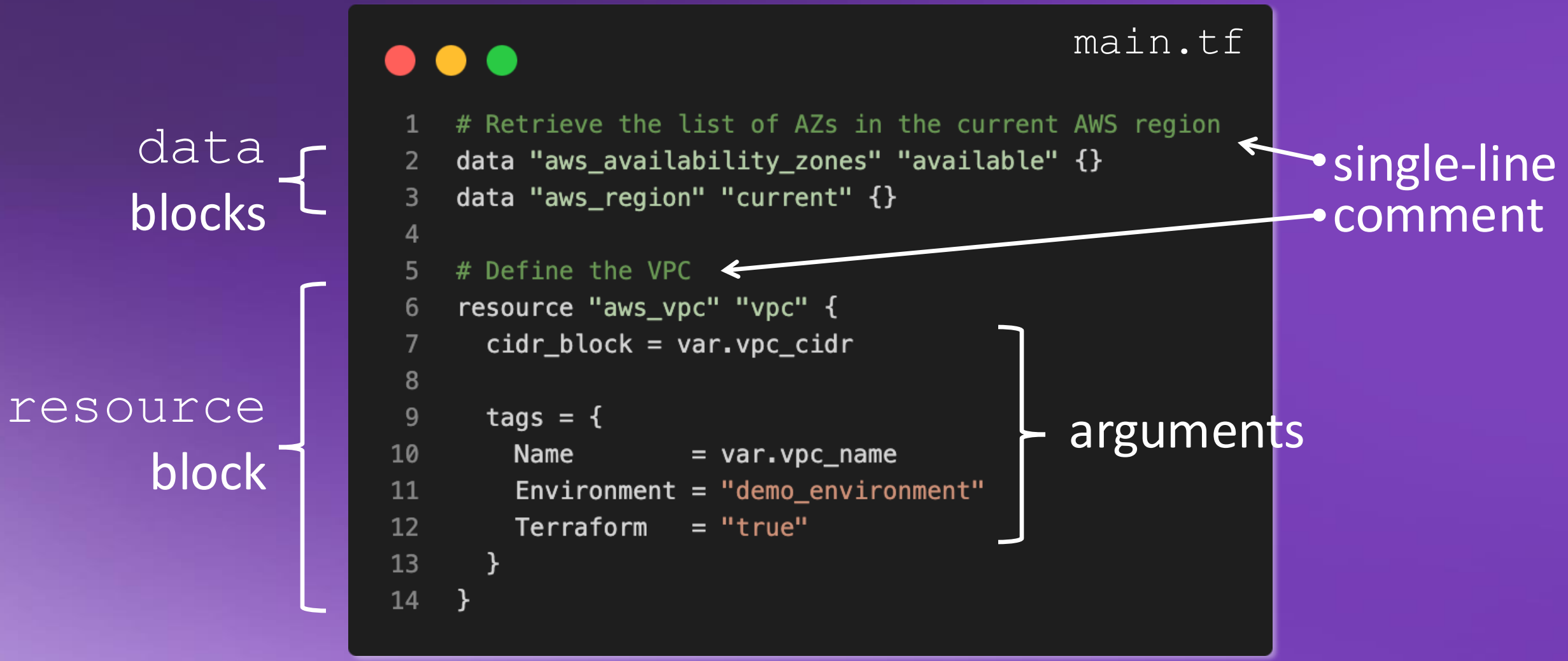
Configuration files that you write tell Terraform what infrastructure you want to create or what data to retrieve

HashiCorp Configuration Language (HCL)

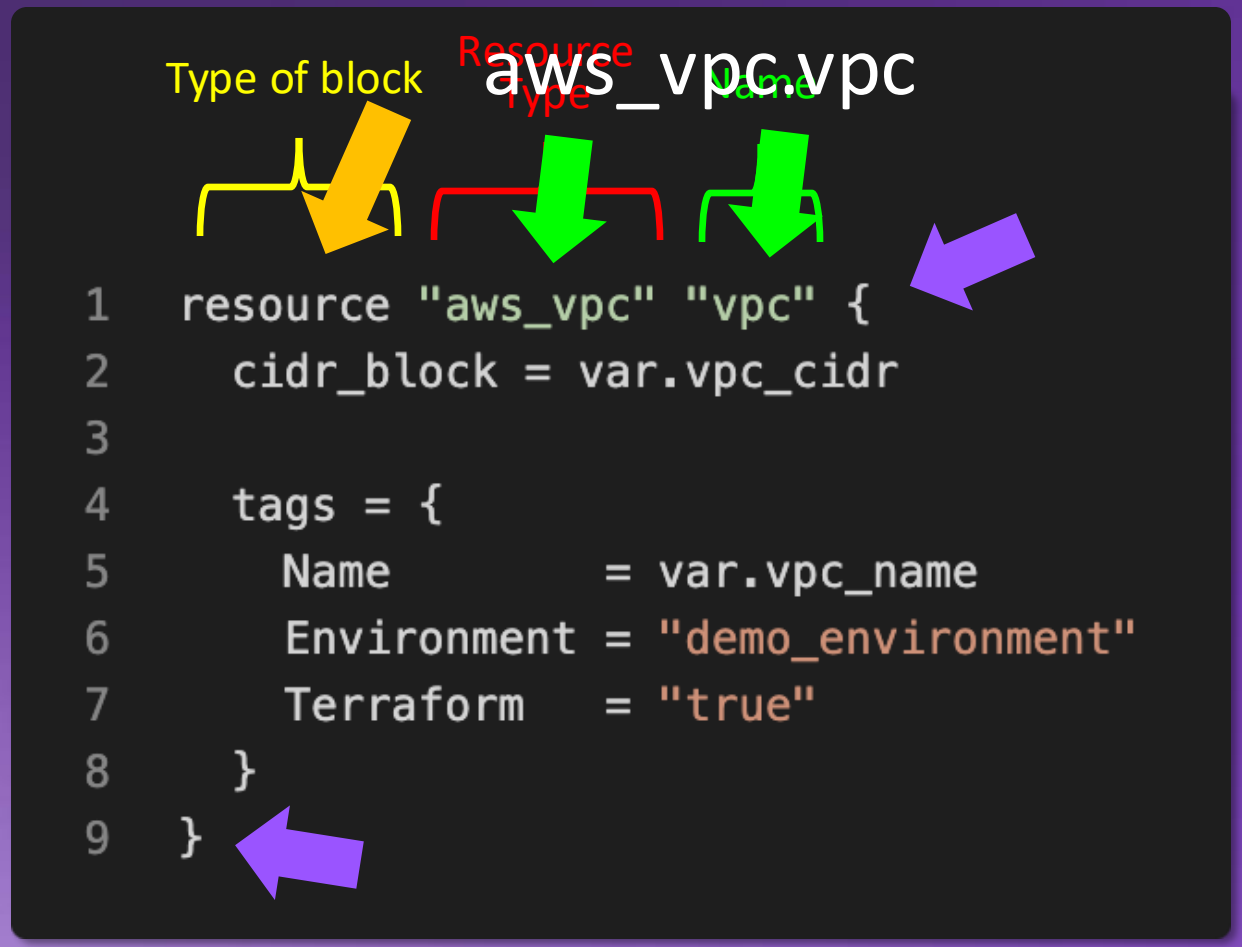


```
1
2  block_type "block_label" "block_label" {
3    first_argument  = expression or value
4    second_argument = expression or value
5    third           = expression or value
6  }
7
8  attribute_abc = "value_1"
9  attribute_2   = "value_2"
```

HCL Example



Block Example



Blocks are delineated by { }
All contents within the block are arguments some type of value

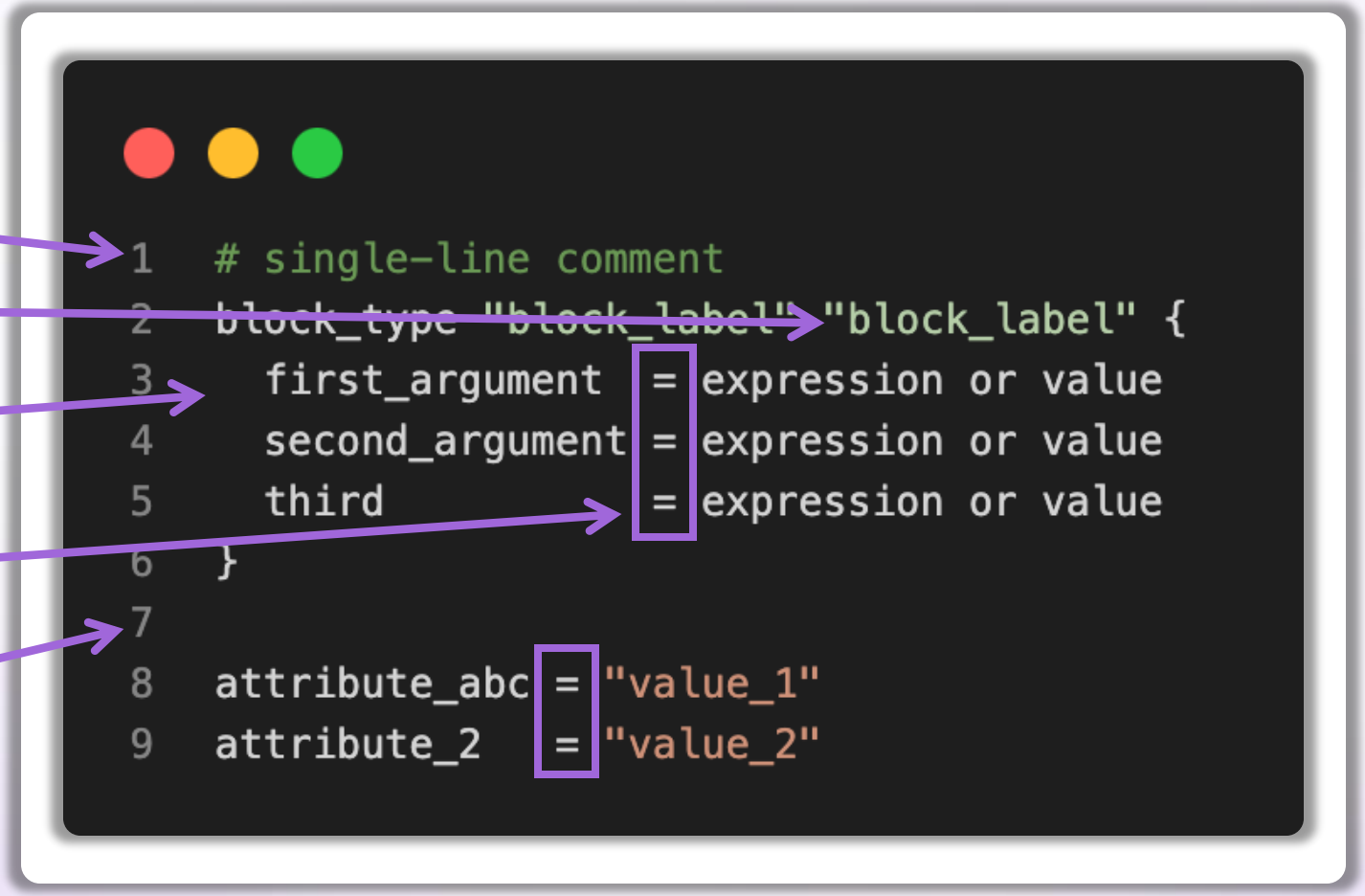
Blocks start with a keyword
Defines what type of block it is, such as resource, data, variable, data, etc.

Blocks have named values
Terraform can access information created by or obtained by a block

HCL Style Guide

Writing Terraform code in a consistent style makes it easier to read and maintain.

- Use # for single and multi-line comments
- Use underscores to separate multiple words in names
- Indent two spaces for each nesting level
- Align equal signs for values on consecutive lines
- Use empty lines to separate groups of arguments in a block



Resource Referencing

Resource referencing allows you to connect resources within your configuration, making it easy to share data between resources and build dependencies.

Create More Dynamic Configurations

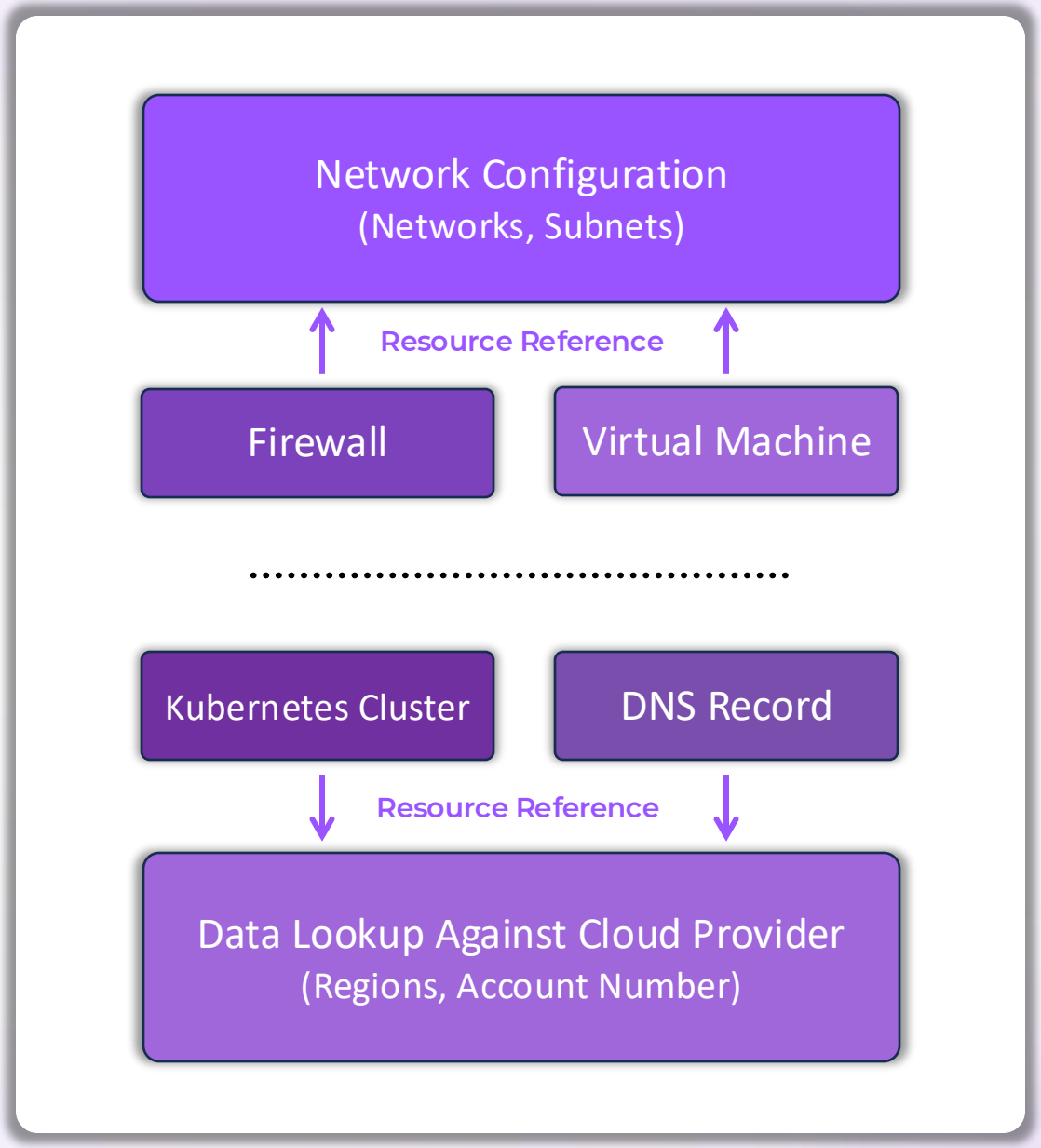
Terraform can use properties from one resource in another and avoid hardcoding values, creating a dynamic configuration

Automatic Dependency Mapping

Terraform automatically determines the order of resource creation based on references.

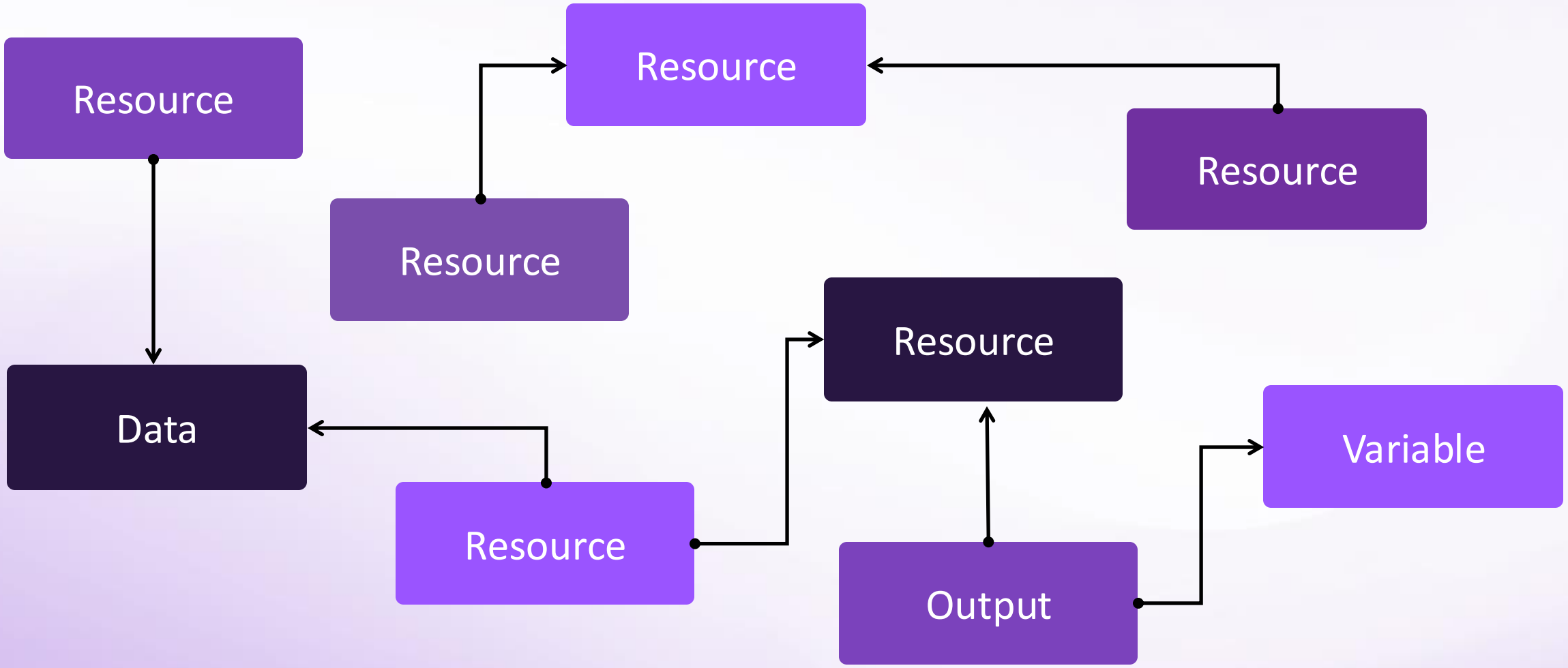
Resource Identifiers

Each Terraform block is identified by a **type** and a **name**, which makes it easy to reference in other parts of your configuration



Resource Referencing

The Reality of Terraform



DEMO

Learn the Basics of HCL

Best Practices for HCL

File Extensions

Terraform HCL files use the .tf .tfvars file extension. This allows Terraform and related tools to recognize and apply configurations properly

Formatting

Correctly formatting files improves readability and maintainability. Follow the style guide or use built-in tools to format your files

Organization

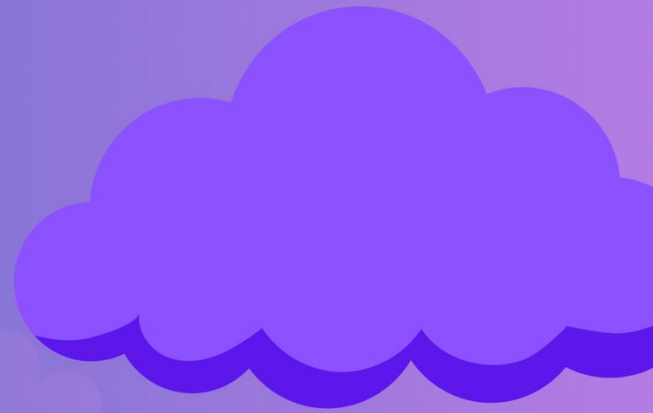
Organize functionality within files (variables in a variables file, outputs in an outputs file, etc.

Commenting and Documentation

Include comments and README.md files to provide context and instructors, making it easier for others to understand and use the code

Avoid Hardcoding Values

Use variables for values that may change which makes configurations for flexible and reuable.

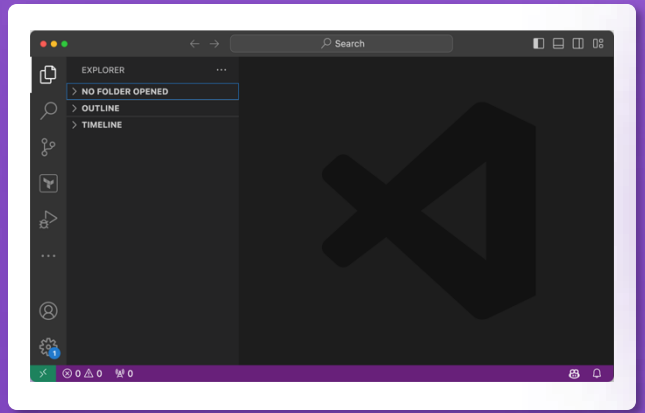


Enhancing Code Quality

Using Linters, Extensions, and Built-In Tools for Code

Properly formatted code improves readability and reduces errors. Use a VSCode extension or linter for real-time syntax checks and best-practice suggestions.

Or run `terraform fmt` to automatically format HCL files with a Terraform-related extension (`.tf`)



Linters and Extensions

Linters provide immediate feedback on code issues, while extensions offer additional features like autocompletion and integrated commands

Built-In Tools

`terraform fmt` automatically formats Terraform code to ensure consistency and readability, making collaboration easier across teams.

SUMMARY

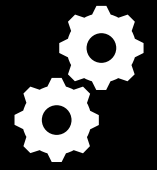
HCL is a declarative language that is easy to read and write, and is Terraform's primary interface.

A single Terraform file can have one or many blocks to define your infrastructure

HCL uses blocks for data, resources, etc., and these blocks can be nested.

Use the '.tf' file extension so Terraform and related tools can recognize and apply configurations properly.

Single-line and block comments are used in HCL.



Use linters, extensions, and built-in tools like 'terraform fmt' to enhance code quality.

LECTURE

Introduction to OpenTofu

HISTORY

2014

First Commit of Terraform Open-Source by Mitchell Hashimoto

2015

Terraform adds state locking for backend

2017

HashiCorp announces Terraform Enterprise.

2018

Terraform 0.12 released, a major upgrade introducing new language syntax improvements

2023

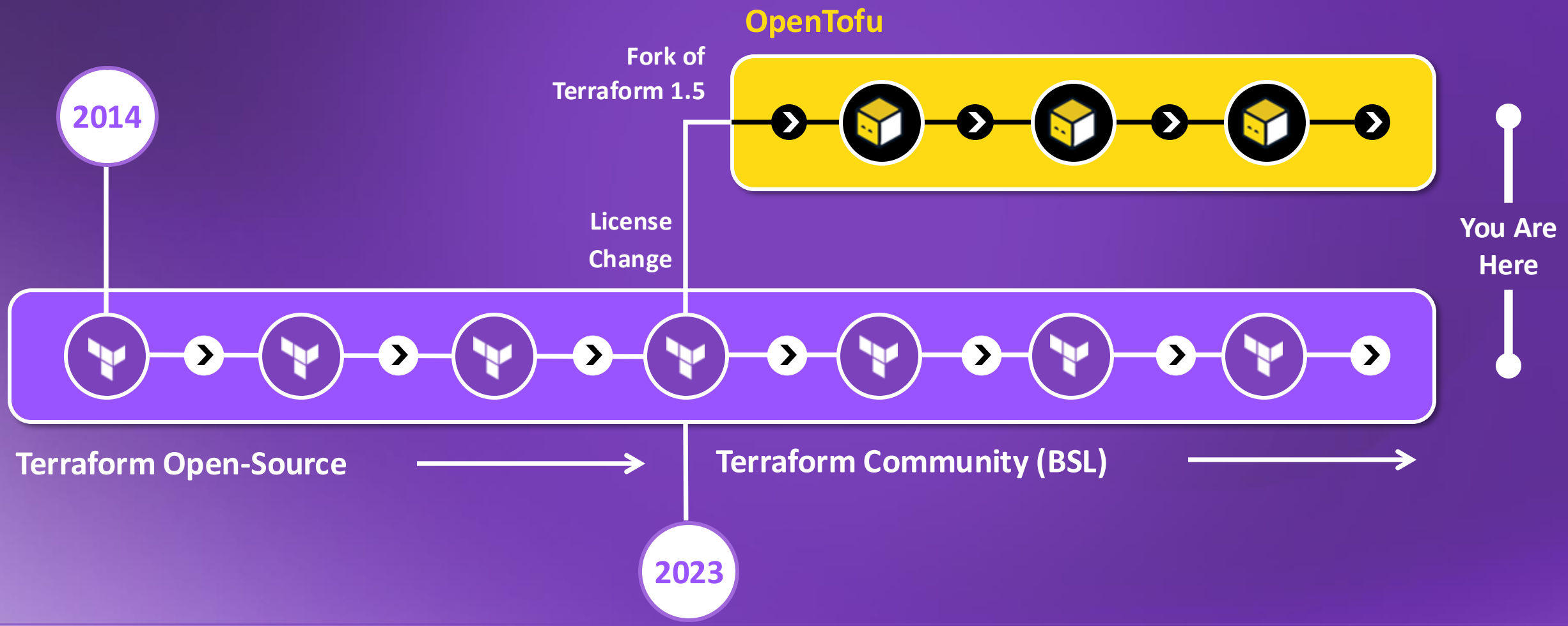
HashiCorp announces a license change from Mozilla to BSL. Terraform no longer "open-source". This leads to the fork of Terraform 1.5 to create OpenTofu

Current State

Terraform and OpenTofu are developed as separate tools.

OpenTofu and Terraform

Understanding the Relationship



Why OpenTofu?

Built from Terraform 1.5, the last MPL version, OpenTofu ensures individuals and organizations retain full access to their infrastructure code without the constraints of proprietary licensing.

Open-Source Commitment

OpenTofu maintains an open-source license (MPL 2.0) to ensure unrestricted use for individuals and organizations.

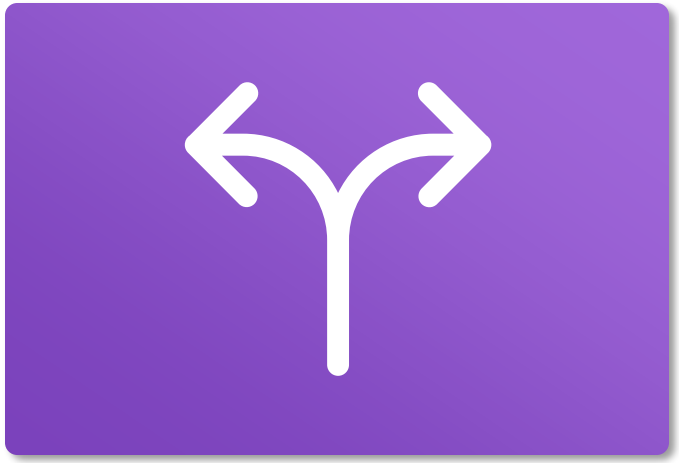
Community-Driven Development

Developers and organizations can influence its roadmap and improvements. Driven by a volunteer steering committee rather than a single company

Seamless Transition for Existing Terraform Users

Enables a smooth migration path for users impacted by the licensing changes while preserving the core Terraform experience.

Differences Between Terraform and OpenTofu

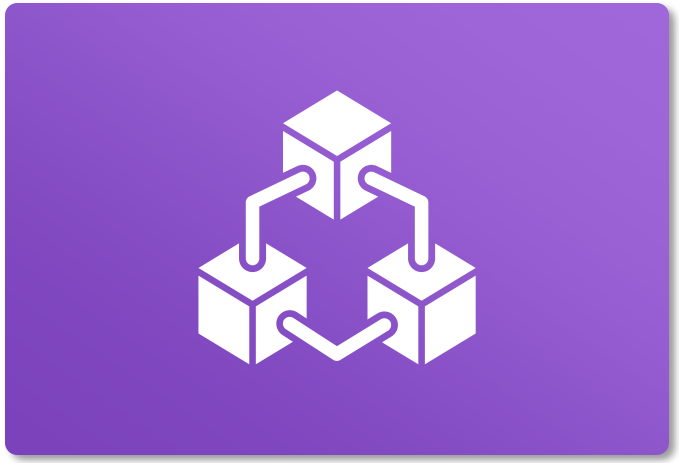


Similar but Different

Since OpenTofu is a fork of Terraform, the differences are subtle (at least for now). OpenTofu supports backward compatibility.

Licensing Concerns

OpenTofu is open-source and managed by the Linux Foundation. Terraform is "source available" but has certain restrictions of use.



New Features

OpenTofu has implemented many long-awaited features that the community has requested for years. HashiCorp is also adding features.

Are Organizations Really Moving to OpenTofu?

Absolutely. Organizations are finding that the features being added to OpenTofu are very impactful to their organization and management of Infrastructure as Code strategy moving forward.

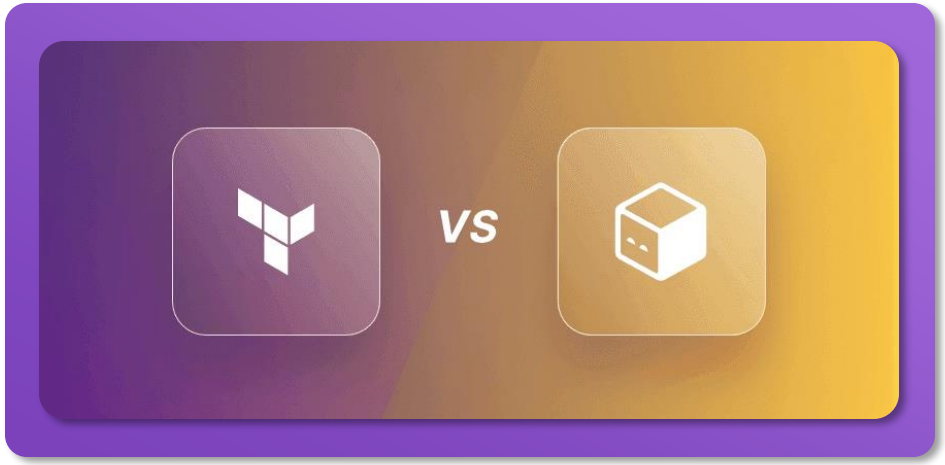
Real-world examples include:

- licensing concerns due to its chosen platform
- support of state encryption
- module source interpolation
- automation platform will not run BUSL versions of Terraform
- preference for licensing which embodies free and open-source software beliefs

Should I Use Terraform or OpenTofu?

Terraform is the de facto standard, but the market is shifting.

As Terraform and OpenTofu continue to introduce unique features at their own pace, the two products are likely to evolve in increasingly different directions. Over time, this divergence could lead to substantial differences in functionality, potentially resulting in limited or no backward compatibility between them in the future.



I Can't Choose for You

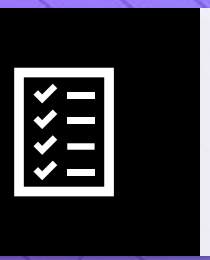
All I can do is give you the best information available, offer my own opinions, and let you make your own decision.

Read: opentofu.org/manifesto

SUMMARY

In 2023, HashiCorp changed the licensing for Terraform, leading to the creation of the open-source fork, OpenTofu.

Licensing concerns, feature advancements, and open-source support are the primary concerns for moving from Terraform



OpenTofu supports backward compatibility with Terraform

Organizations have moved to OpenTofu, but many others still use Terraform daily.

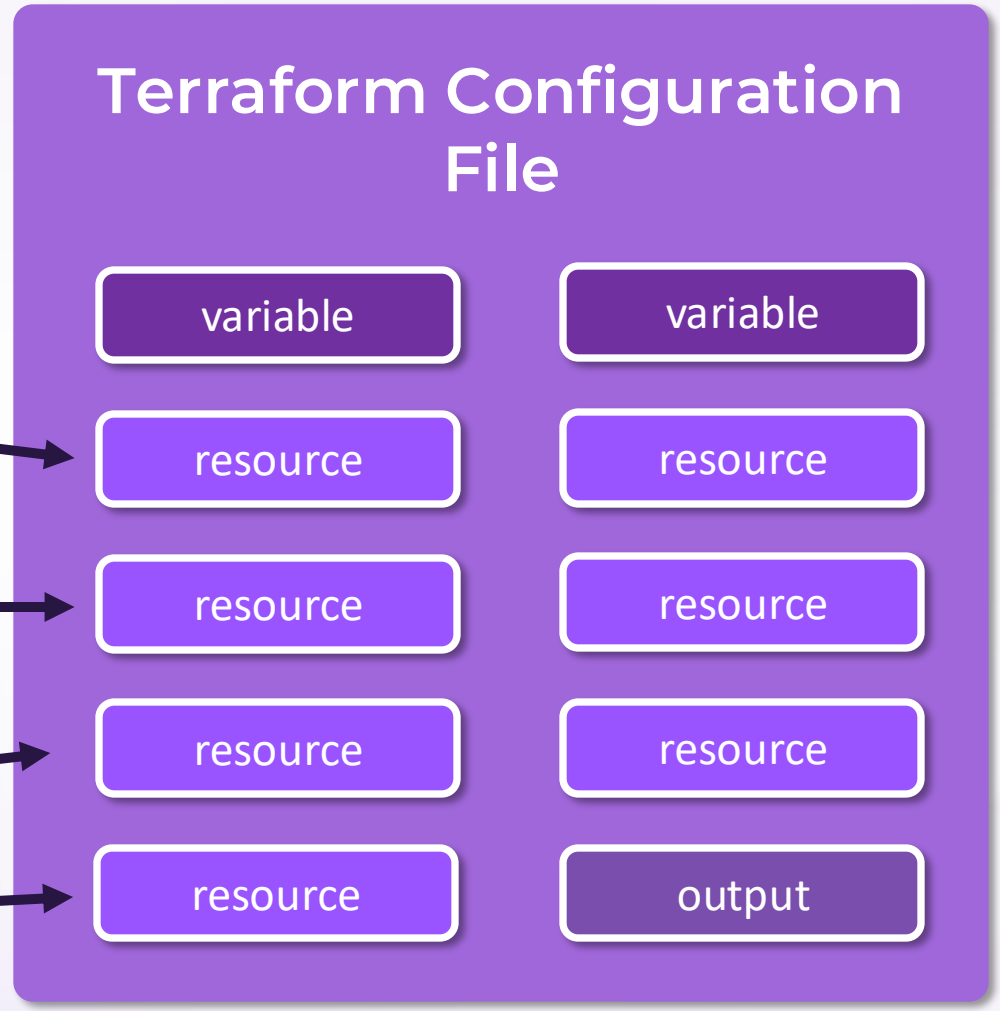
LECTURE

File Structure and Organization

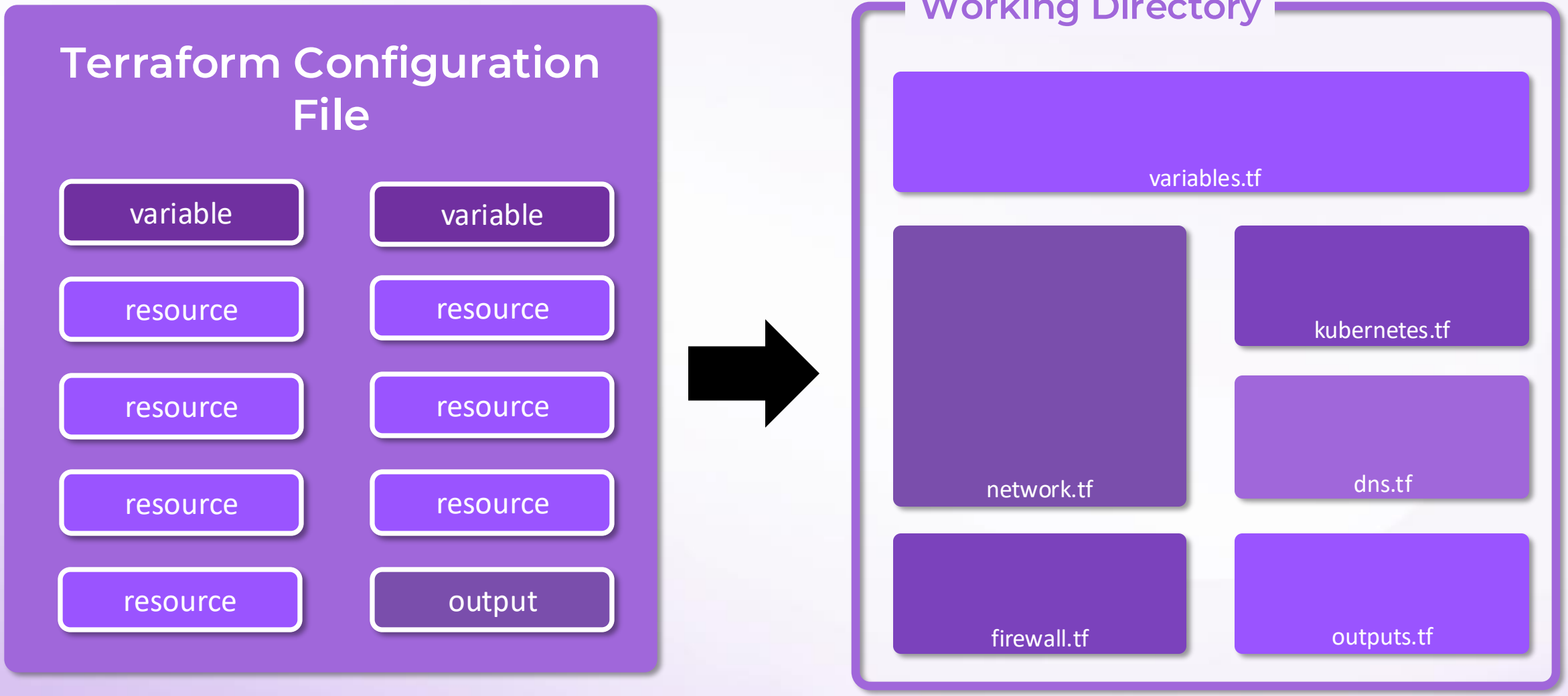
Organizing Code

Infrastructure Requirements:

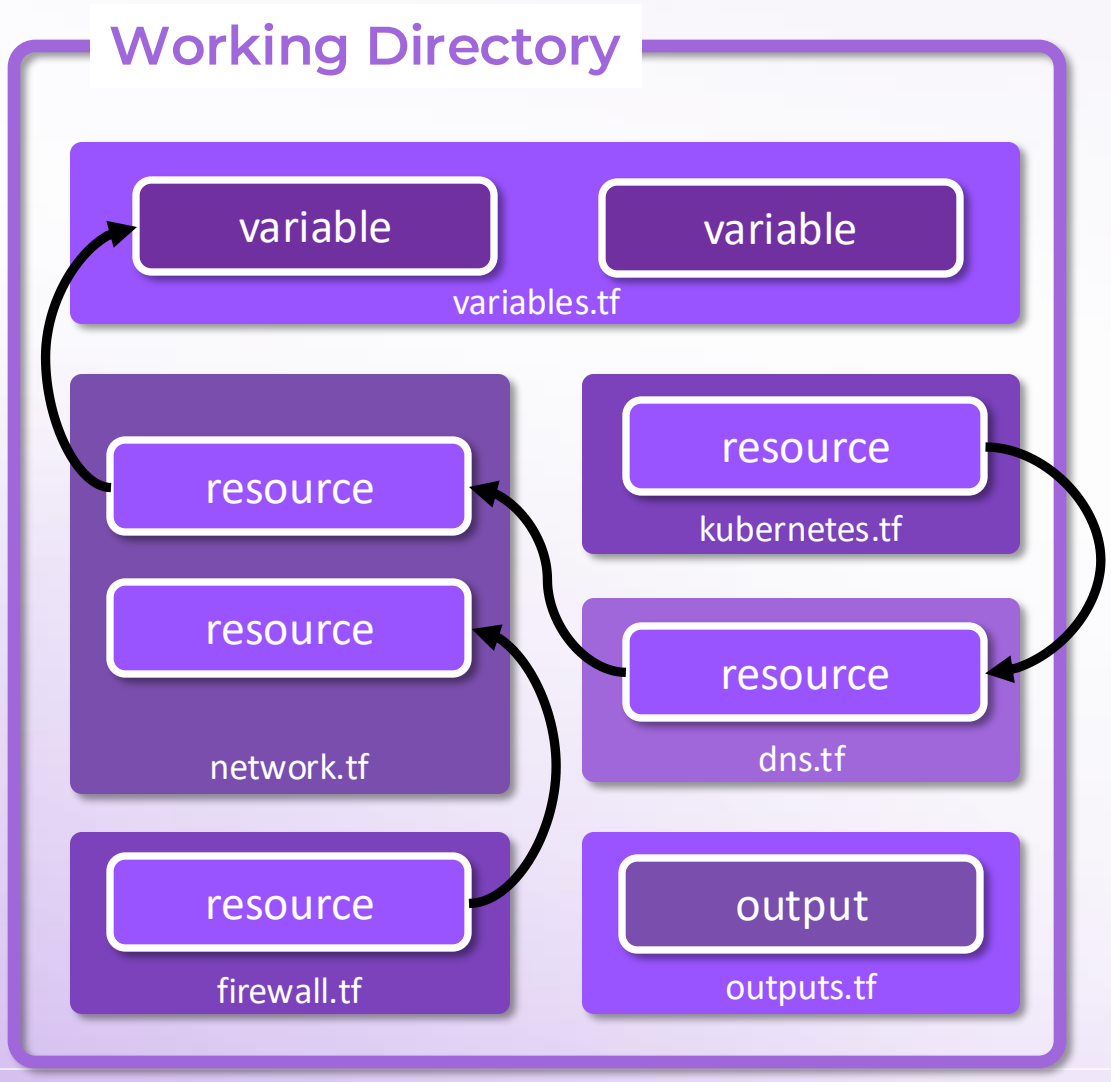
- ❑ S3 Bucket
- ❑ Virtual Machine
- ❑ Subnet
- ❑ Firewall Configuration
- ❑ DNS Record
- ❑ Load Balancer
- ❑ Kubernetes Cluster



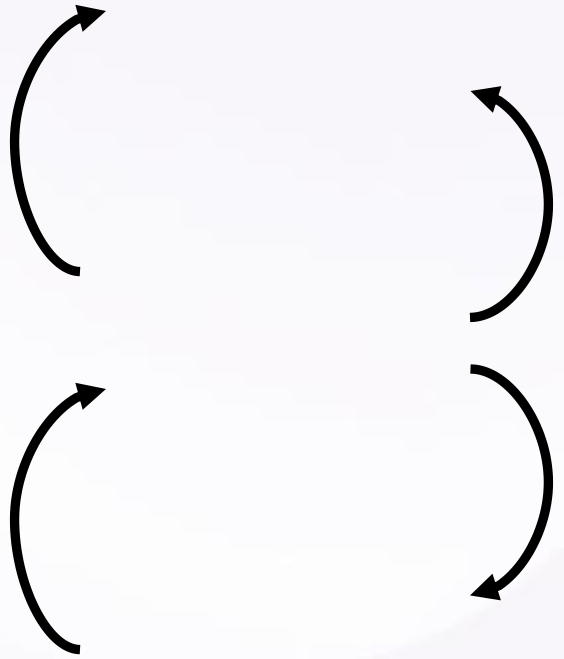
Organizing Code



Executing Code



Terraform Processes All Files Together



Common Terraform Files

main.tf
Commonly used for primary infrastructure components

variables.tf
Variable definitions used throughout your infrastructure

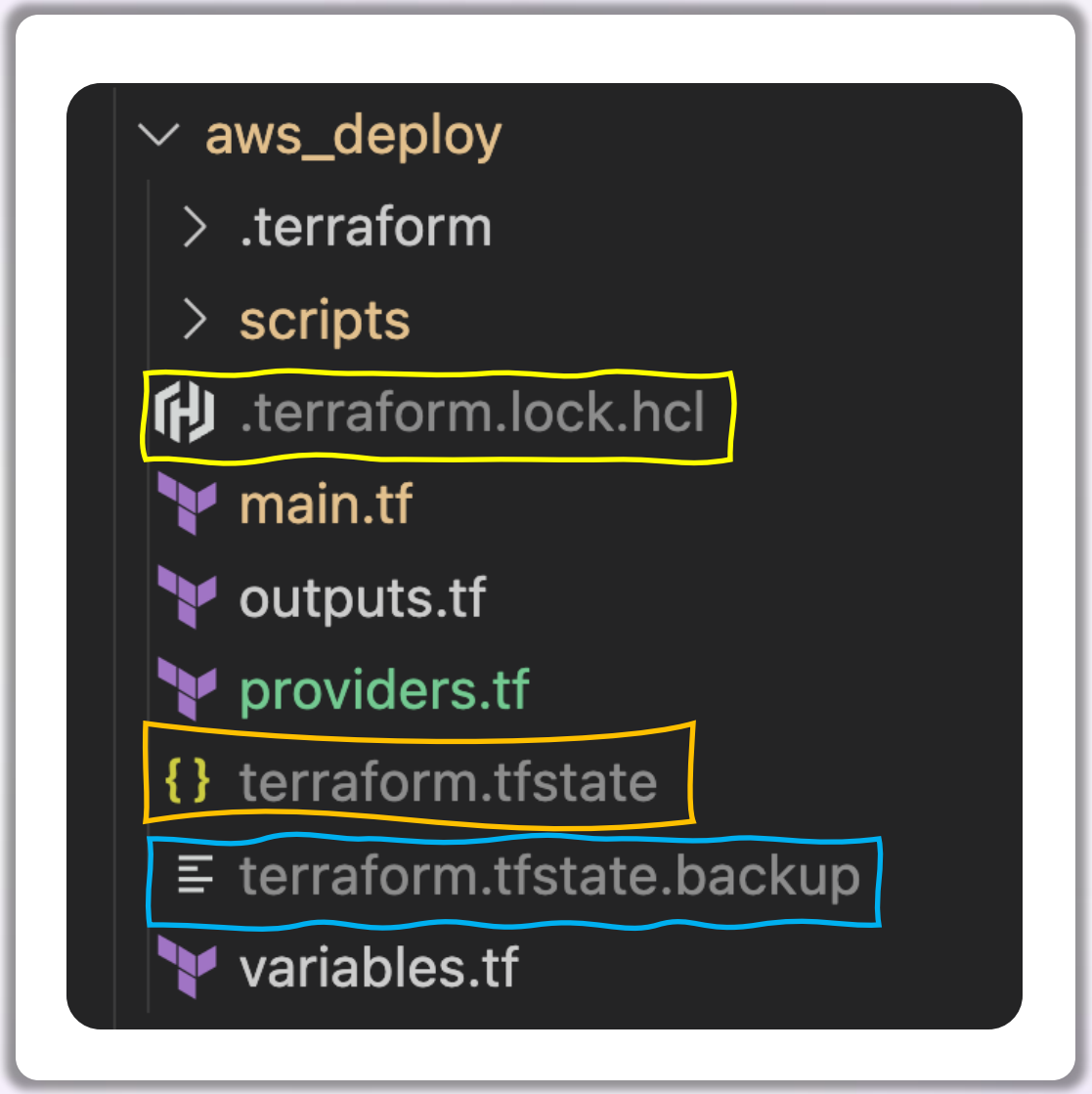
outputs.tf
Output blocks to output values to the terminal or to use for inputs to other modules

providers.tf
Provider configurations and requirements

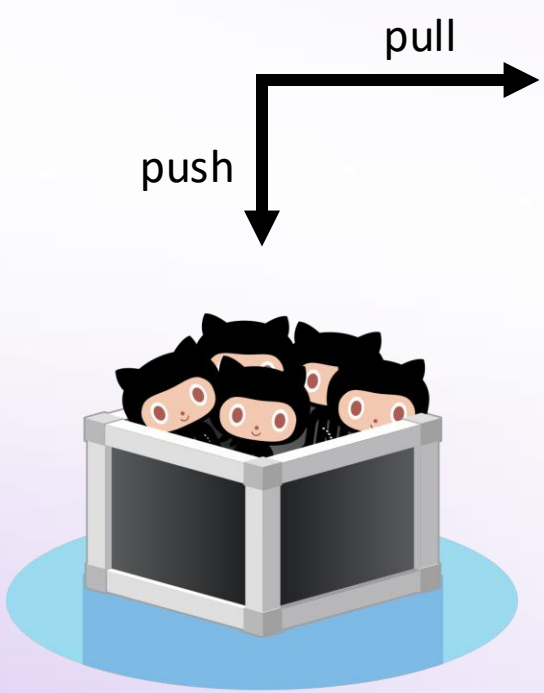
terraform.tfvars
Define the values for your variables – usually ignored by .gitignore

Additional Files

- `terraform.tfstate`
  The file that Terraform uses to store state
- `terraform.tfstate.backup`
  Backup of the previous state file before Terraform writes anything new to the state file
- `.terraform.lock.hcl`
  Used to track and select provider versions
- `.gitignore`
  Instructs Git to ignore files included in this file



Working Directory



Git Repo

pull

push

terraform

prod

test

dev

projectX

main.tf

variables.tf

outputs.tf

state

main.tf

variables.tf

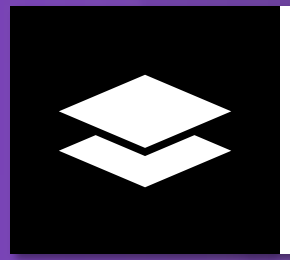
outputs.tf

state

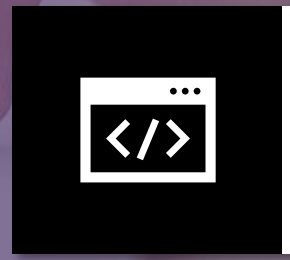
SUMMARY

Terraform only processes the files in the current working directory unless you reference other files

While not required, there are common file names for Terraform configurations

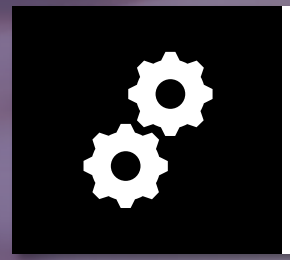


You can organize your files across multiple .tf files



Terraform creates and manages files for state and other configurations

Terraform combines all the files in the working directory when executing



Use subdirectories to break up your Terraform to better organize and reduce the blast radius of changes

LECTURE

Understanding the Terraform Blocks

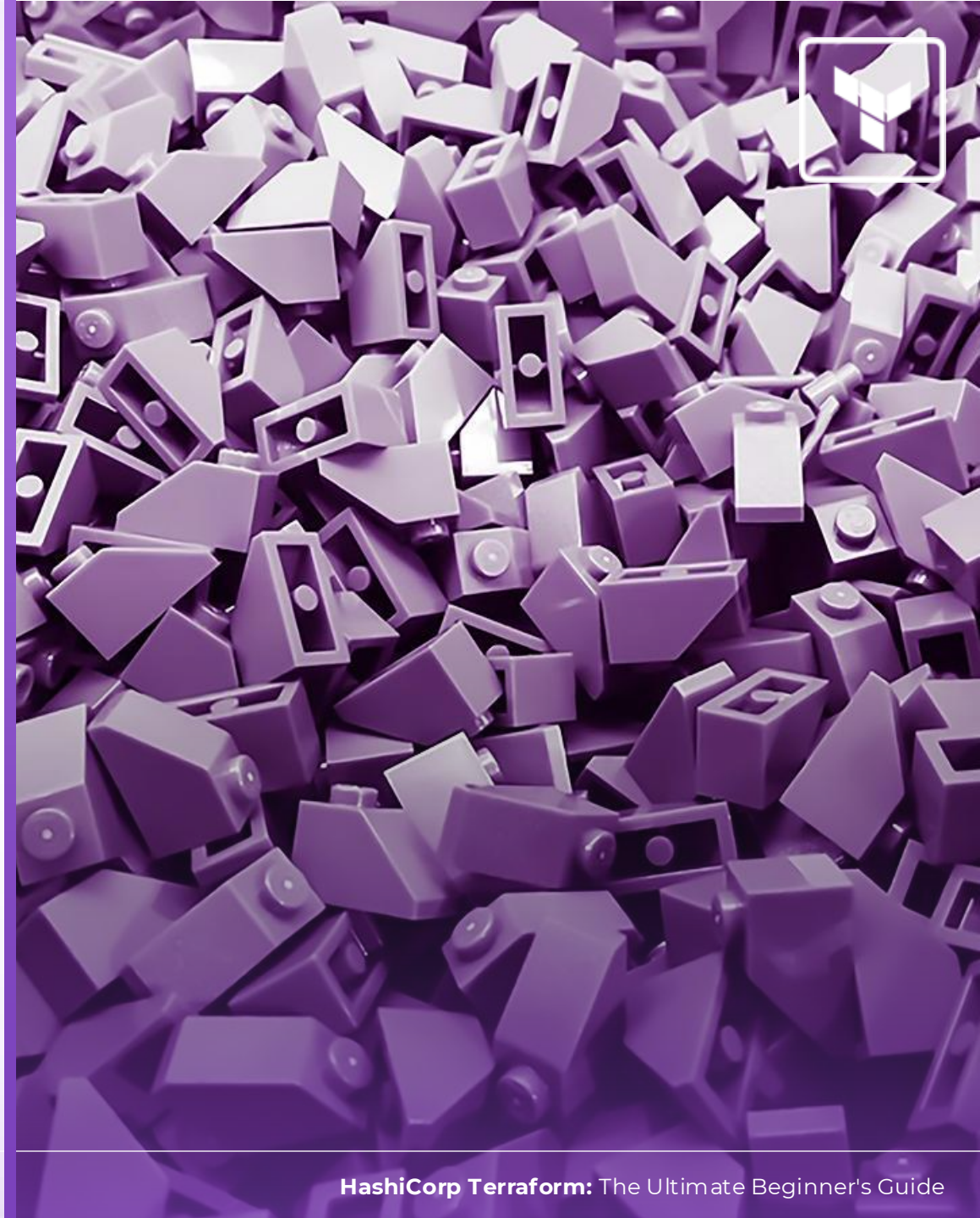
Terraform Blocks

The Building Blocks or Infrastructure

Clear Structure
Each component has a specific purpose, making it easy to organization and understand your configurations

Flexible and Modular
Terraform is designed to be flexible. You don't need to use all the components to build infrastructure

Reusable Patterns
Using components thoughtfully helps create reusable, modular configurations that are easy to maintain and scale



Terraform Block Types

provider

Connect Terraform to cloud platforms or services (e.g., AWS, Azure, GCP, VMware)

resource

The infrastructure that Terraform creates, updates, or deletes (e.g., VMs, storage)

data

Retrieve information about existing resources to use in your configuration

variable

Make code reusable by defining values that can be customized for each run

output

Display or share essential information about resources after deployment

terraform

Define specific settings or resources in a configuration, like req. Terraform version

module

Reusable configurations that group related resources – makes it easy to share configs

import

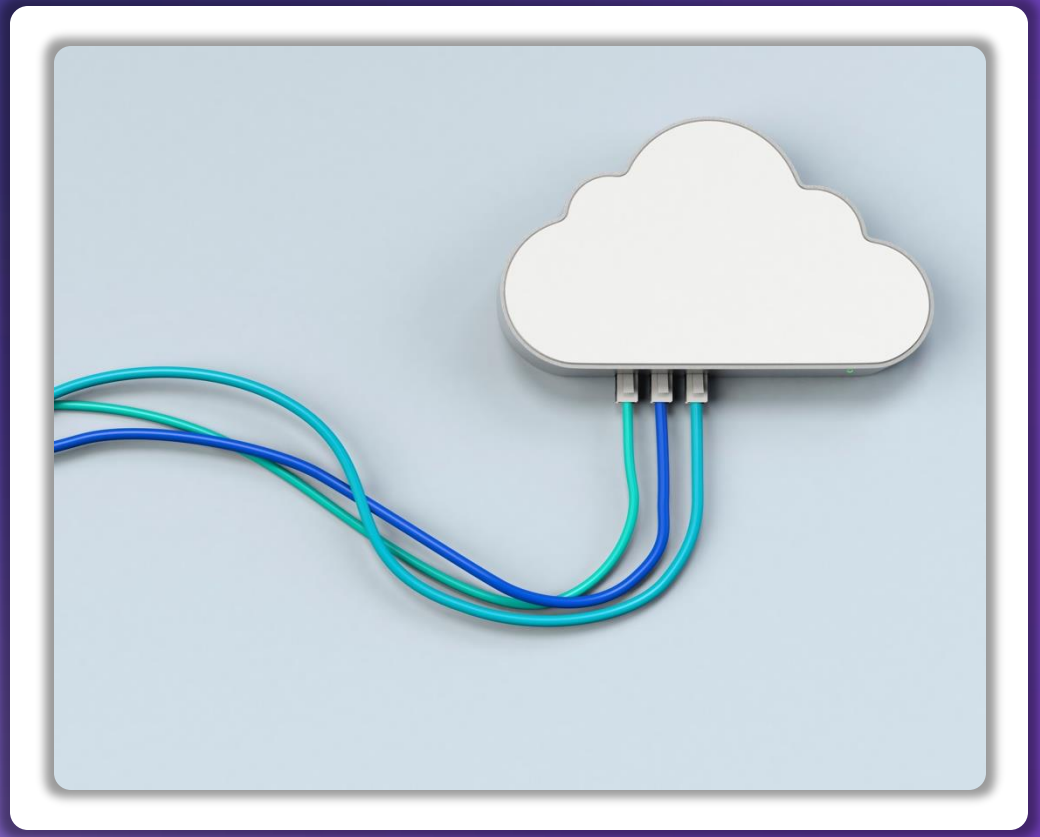
Pull in existing resources into Terraform management

***not an exhaustive list of block types**

LECTURE

Provider Block

Terraform Providers



Provider Block

The `provider` **block** is used to extend Terraform's functionality. It allows you to connect Terraform to any API-driven platform, such as cloud platforms, SaaS applications, on-premises hardware solutions, and more.

Provider blocks define where and what Terraform can manage, such as AWS for cloud resources or GitHub for repositories.

Provider Block

Plugins that extend the functionality of Terraform and enable it to interact with cloud providers, SaaS providers, and other API-supported apps.

- **Resources are Implemented by a Provider**

  Without providers, Terraform would not be able to provision or manage any infrastructure.

- **Who Publishes Providers?**

  HashiCorp maintains some providers, but some are managed by partners and the Terraform community.

- **Developed Separately From Terraform Core**

  This means functionality can be added/changed per version.

providers.tf

```
provider "aws" {
  region  = "us-east-2"
  profile = "prd-workload"
}


provider "azurerm" {
  features {}

  tenant_id       = "tenant-id"
  subscription_id = "sub-id"
  client_id       = "client-id"
  client_secret   = "client-secret"
}
```

Breaking Down the Provider Block

providers.tf

```
provider " aws" {
  region  = "us-east-2"
  profile = "prd-workload"
}

provider "azurerm" {
  features {}

  tenant_id       = "tenant-id"
  subscription_id = "sub-id"
  client_id       = "client-id"
  client_secret   = "secret"
}
```

The name of the provider defined by the developer or maintainer

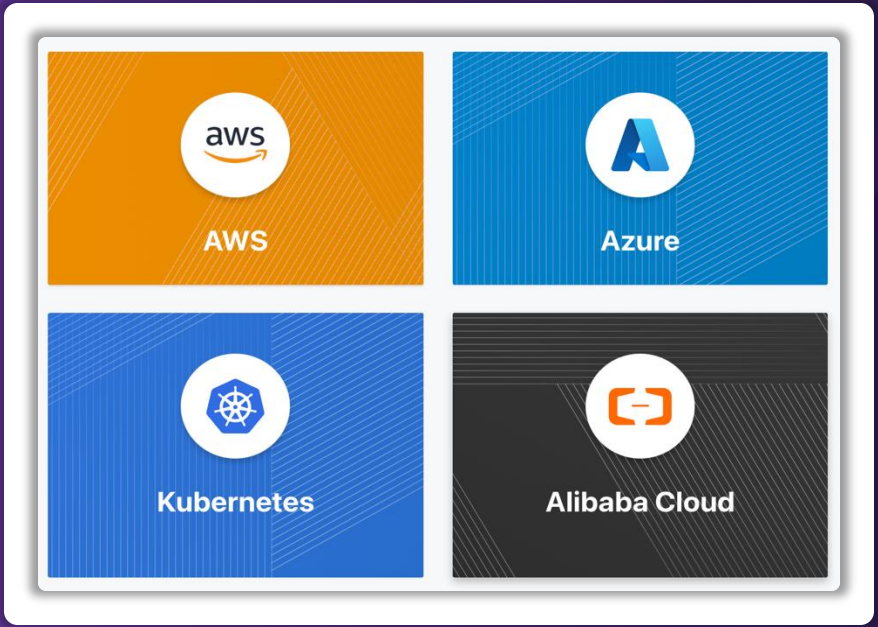
Arguments specific to the provider – found in documentation

Authentication arguments – should be provided via Environment Variables instead due to security concerns

Provider Block

- Some providers need credentials or access tokens to authenticate and manage resources on the target platform (e.g., AWS access keys, GitHub API token)
- A single provider block enables Terraform to manage multiple resources on that platform (e.g., a single AWS provider enables EC2 instances, S3 buckets, EKS clusters)

Provider Block

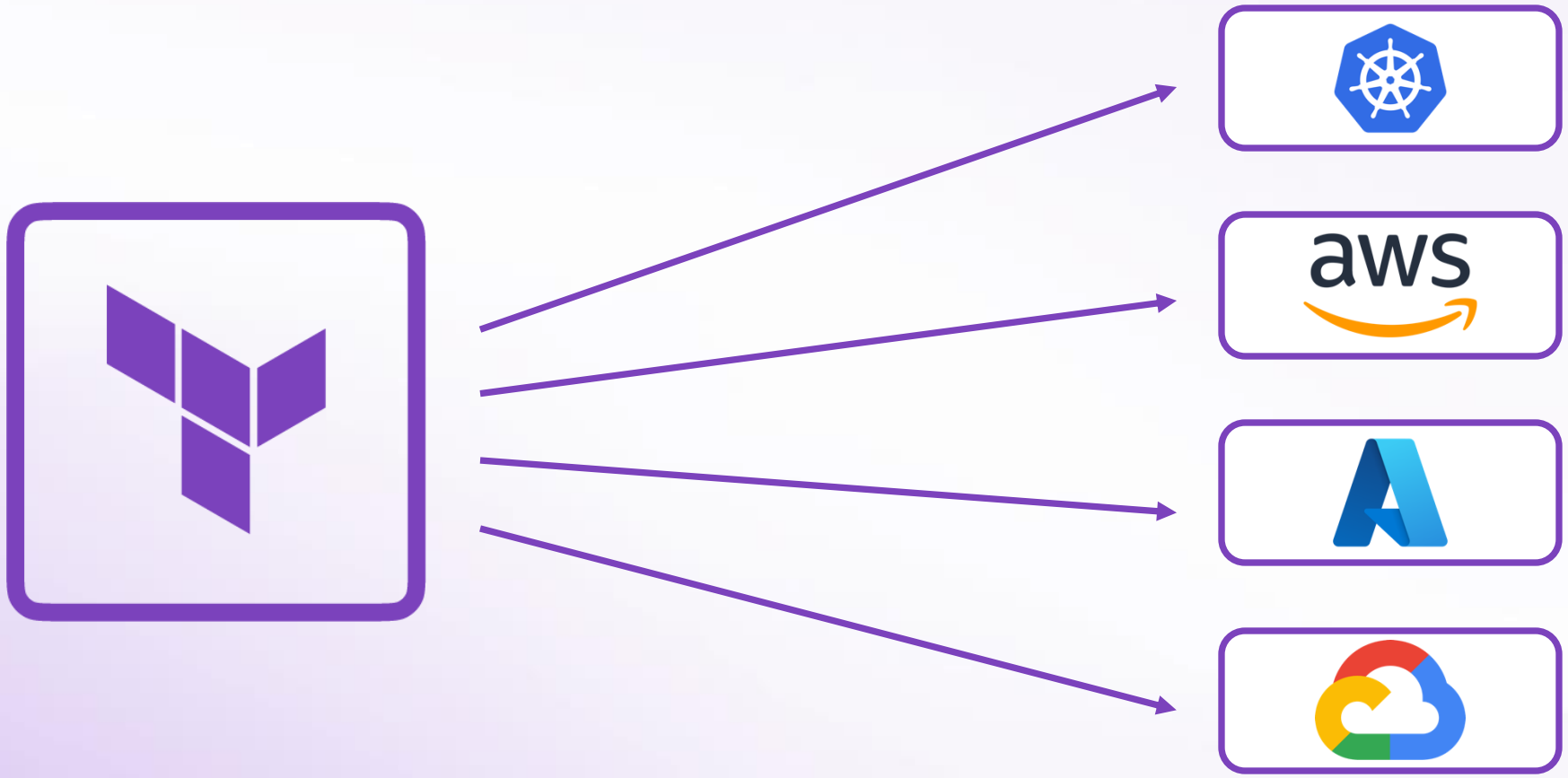


- While providers are stored in GitHub, the documentation can be found on the Terraform Registry → registry.terraform.io
- When declared in your configuration, the provider is downloaded to the machine executing Terraform

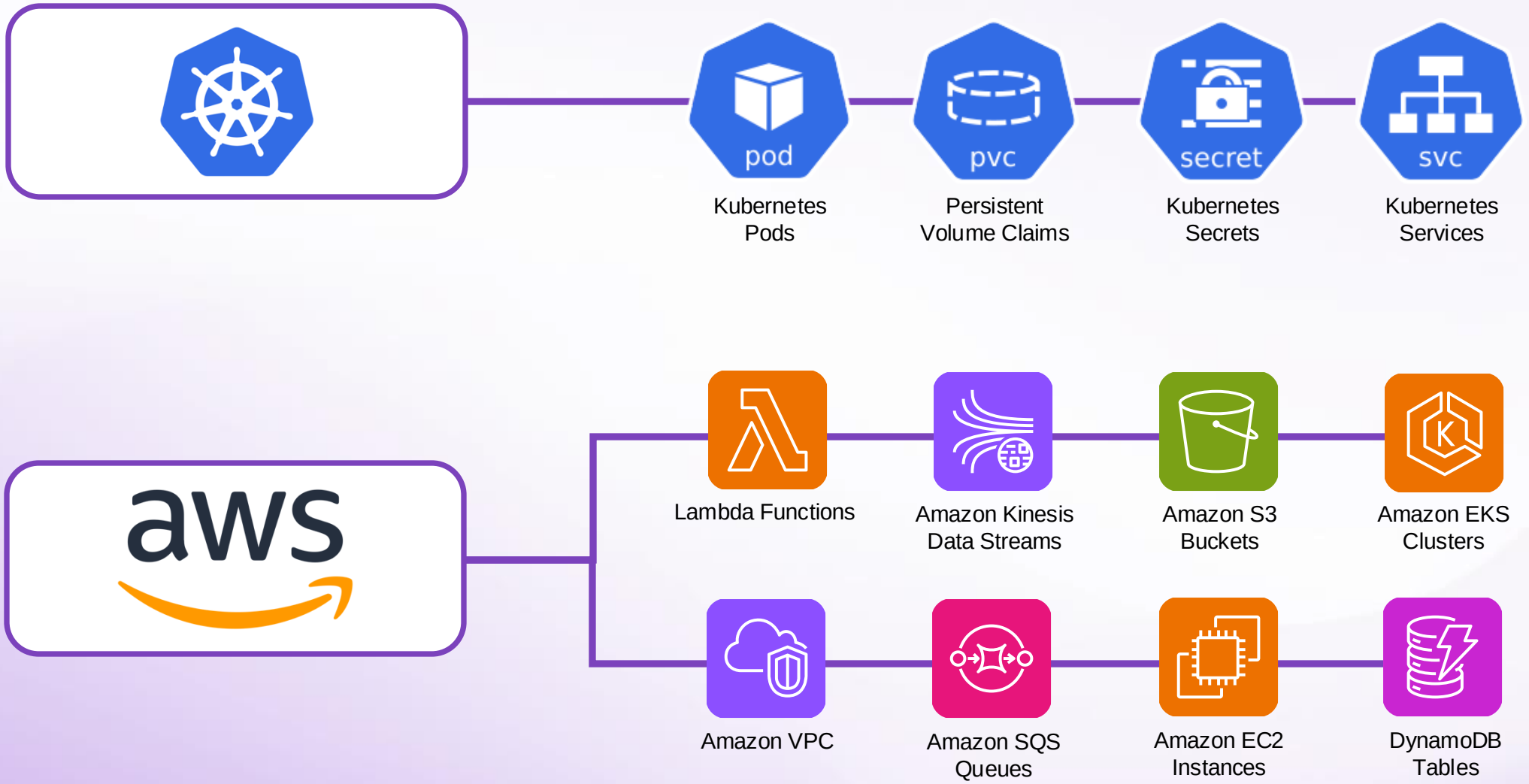
LECTURE

Resource Block

Terraform Resources



Terraform Resources



Resource Block

The `resource` **block** is the core element in Terraform. It defines the specific infrastructure pieces you want to create, manage, or modify, such as virtual machines, databases, or networks.

Resource blocks use a resource type (e.g., `aws_instance`) and a unique name. It acts as a blueprint with the settings you need to configure each resource.

Resource Block

A resource block tells Terraform what to build and how it should be configured.

What Type of Resource is Needed?

The resource type identifies the type of resource that you need. Value is determined by the provider depending on the desired resource.

How Should the Resource be Configured?

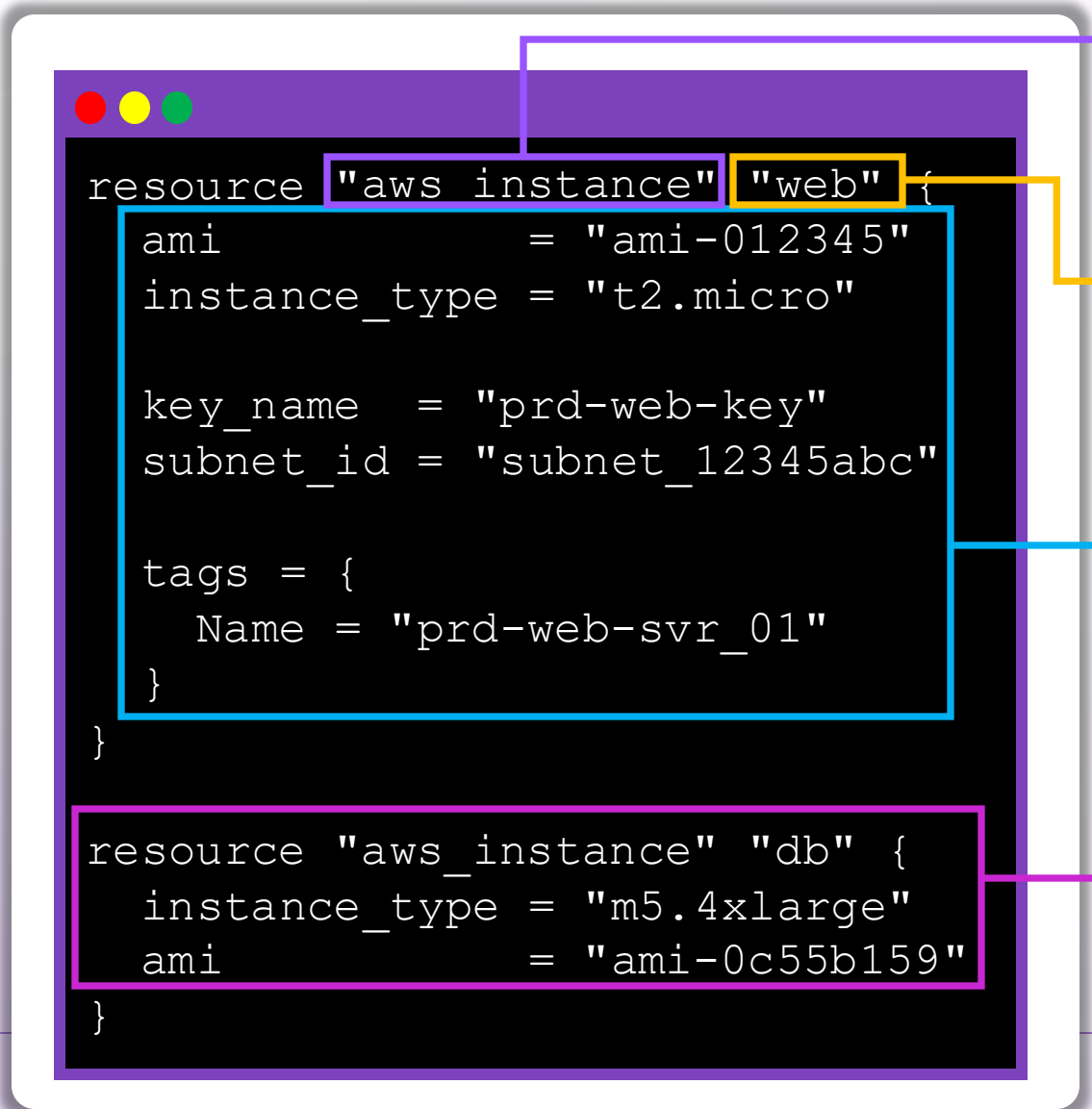
Each resource block includes settings (or arguments) that configure its behavior.

Resource Identifiers

Each resource block is identified by a **type** and a **name**, which makes it easy to reference in other parts of your configuration

```
resource "aws_instance" "web" {
  ami           = "ami-012345"
  instance_type = "t2.micro"

  key_name  = "prd-web-key"
  subnet_id = "subnet_12345abc"

  tags = {
    Name = "prd-web-svr_01"
  }
}

resource "aws_instance" "db" {
  instance_type = "m5.4xlarge"
  ami           = "ami-0c55b159"
}
```

Breaking Down the Resource Block



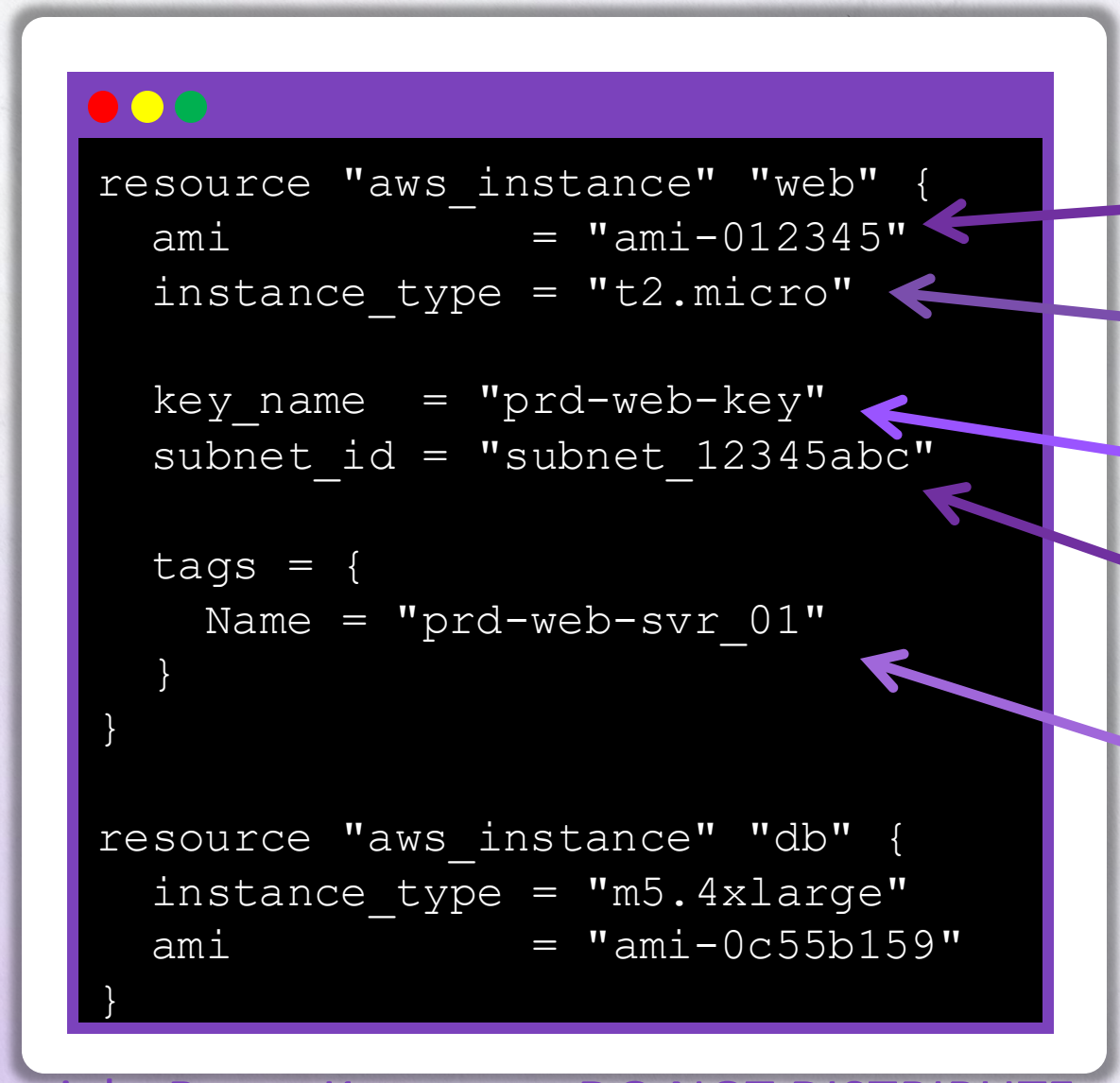
The type of resource to be created/ managed - defined within the provider

The resource name – user-provided string - must be unique per configuration.

Arguments specific to the resource type Can be found in provider documentation

A second resource block within the same Terraform configuration file

Resource Block

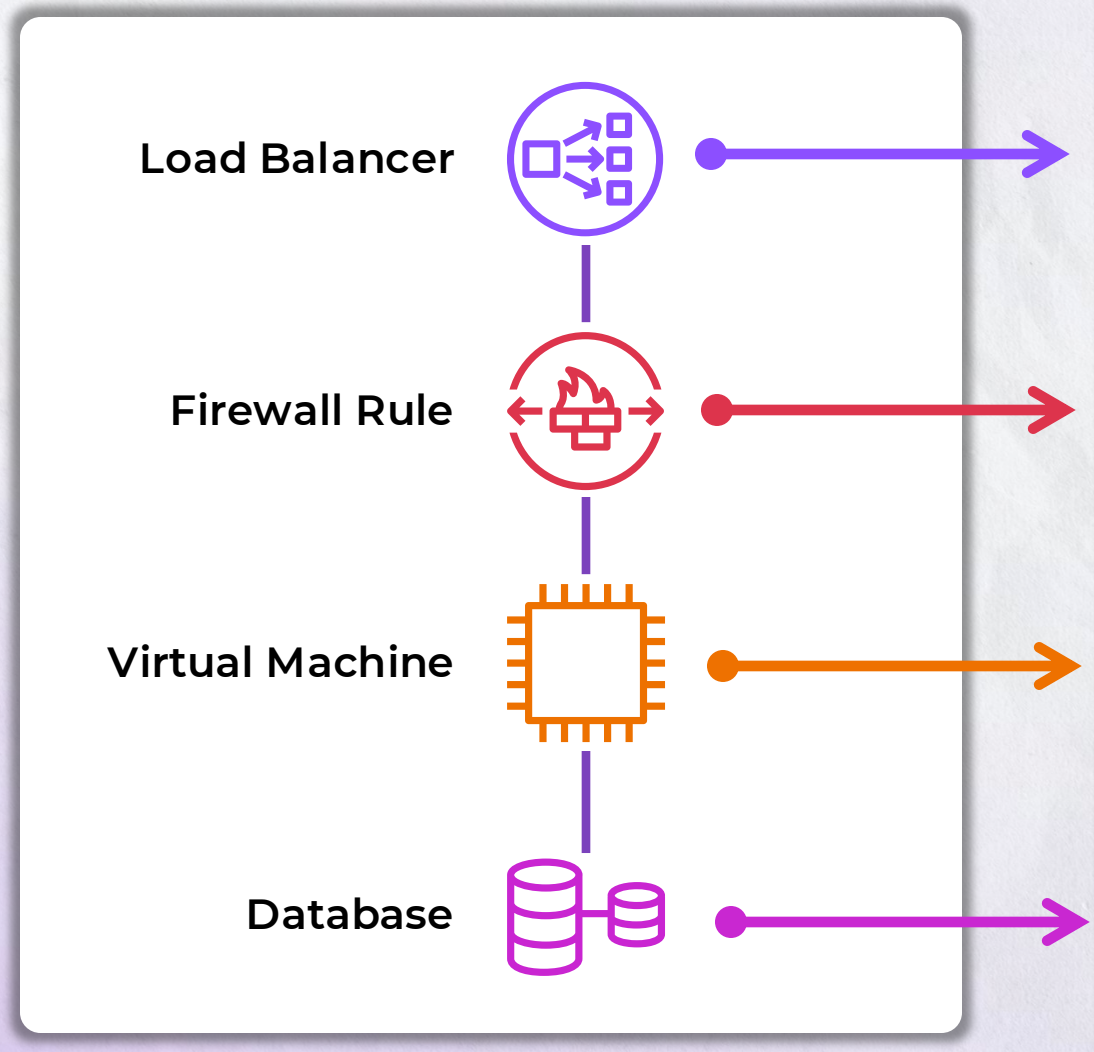


```
resource "aws_instance" "web" {
  ami           = "ami-012345"
  instance_type = "t2.micro"

  key_name  = "prd-web-key"
  subnet_id = "subnet_12345abc"

  tags = {
    Name = "prd-web-svr_01"
  }
}

resource "aws_instance" "db" {
  instance_type = "m5.4xlarge"
  ami           = "ami-0c55b159"
}
```

Resource Arguments

- The Image to Use
- The Size of the Instance
- Key Name for SSH Access
- The Subnet used to Provision the Instance
- Add Tags for the Resource

Resource Block



web_app.tf

```
resource "aws_lb" "public" {
  name = "prd-web-lb"
  load_balancer_type = "network"
}

resource "fortios_firewall_policy" "db" {
  action = "accept"
  name   = "allow_web_443"
}

resource "aws_instance" "web" {
  instance_type = "t3.large"
  ami           = "ami-0c55b159"
}

resource "aws_db_instance" "prd-db" {
  engine         = "mysql"
  instance_class = "db.t3.large"
}
```

Resource Referencing

Dynamically pull information from one resource block to another, creating a connected configuration

Reference a Resource block by using the resource **type** and the **name** of the resource.

1. `aws_lb.public_load_balancer`
2. `fortios_firewall_policy.db`
3. `aws_instance.web`
4. `aws_db_instance.prd_db`

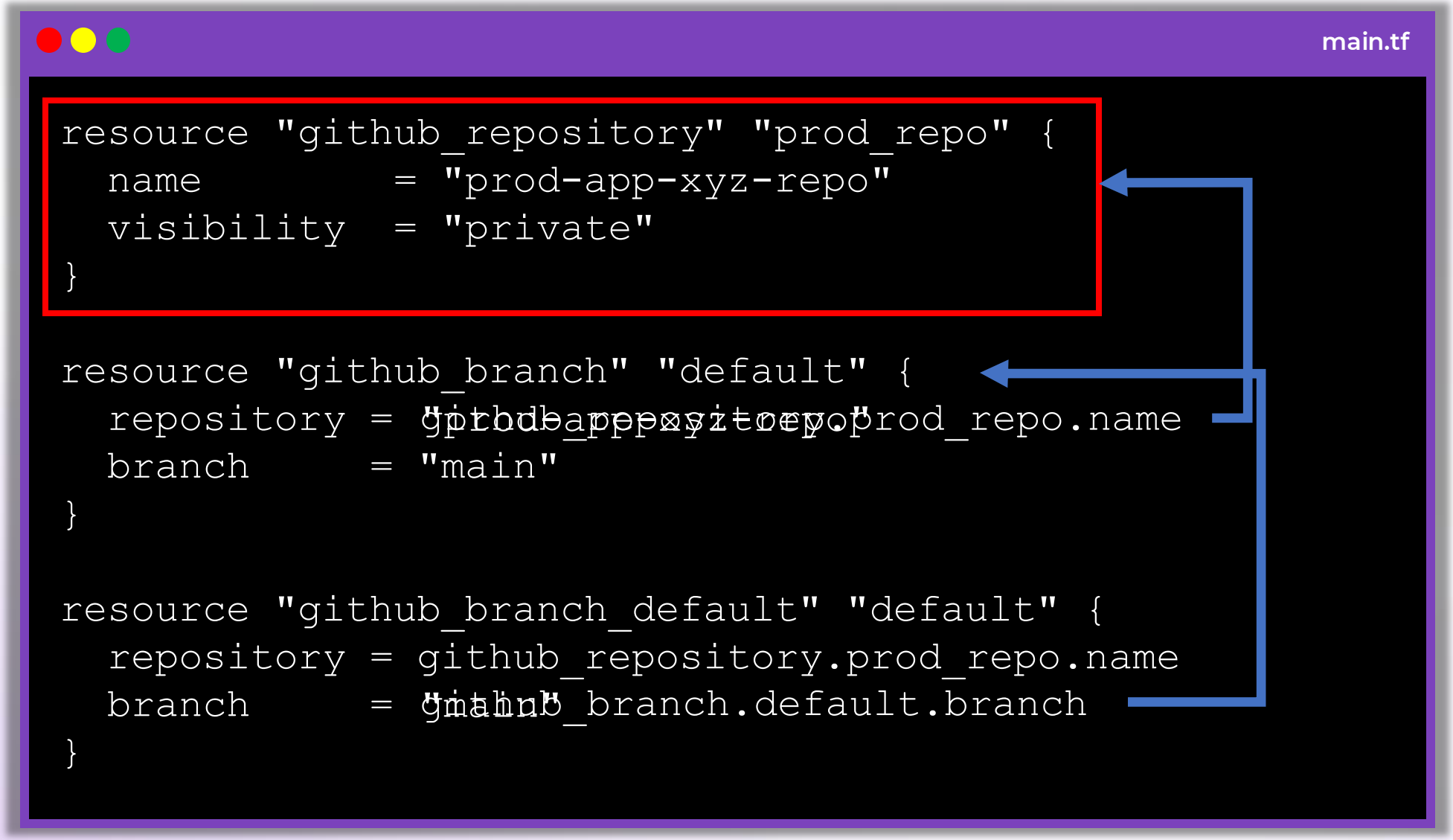
web_app.tf

```
1  resource "aws_lb" "public_load_balancer" {
     name = "prd-web-lb"
     load_balancer_type = "network"
   }

2  resource "fortios_firewall_policy" "db" {
     action = "accept"
     name   = "allow_web_443"
   }

3  resource "aws_instance" "web" {
     instance_type = "t3.large"
     ami           = "ami-0c55b159"
   }

4  resource "aws_db_instance" "prd_db" {
     engine         = "mysql"
     instance_class = "db.t3.large"
   }
```

Resource Block

main.tf

```
resource "github_repository" "prod_repo" {
  name         = "prod-app-xyz-repo"
  visibility   = "private"
}

resource "github_branch" "default" {
  repository = github_repository.prod_repo.name
  branch     = "main"
}

resource "github_branch_default" "default" {
  repository = github_repository.prod_repo.name
  branch     = github_branch.default.branch
}
```



Resource Block

- Resource names must start with a letter or underscore, and may contain only letters, digits, underscores, and dashes.

HashiCorp Terraform: The Ultimate Beginner's Guide

LECTURE

Data Block

Accessing Existing Resources

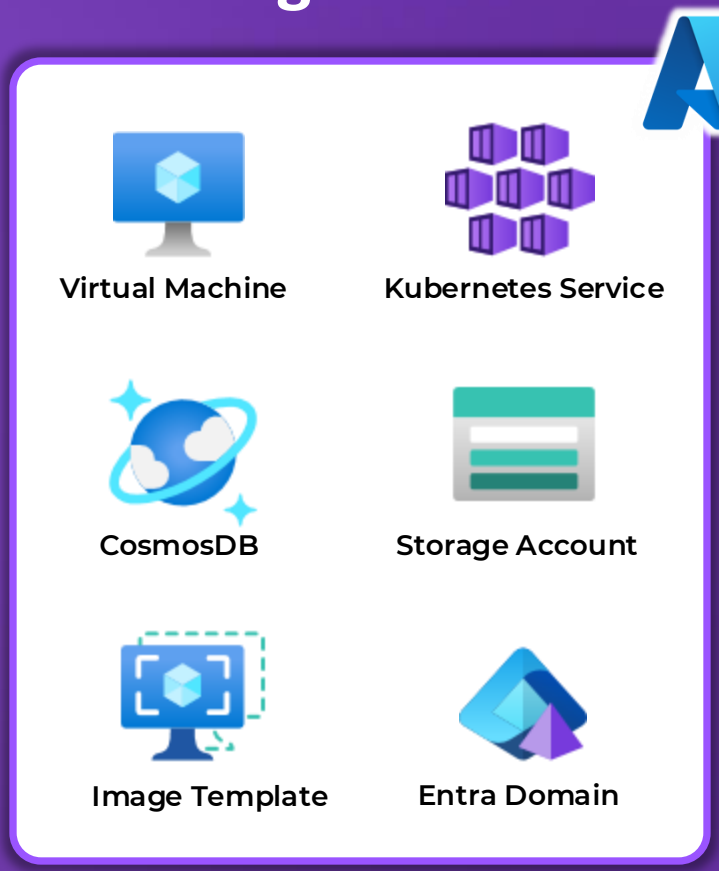
```
resource "azurerm_resource_group" "prd" {
  name     = "example-resources"
  location = "West Europe"
}

resource "azurerm_virtual_network" "dv" {
  name                = "example-network"
  resource_group_name = "azure"
  location            = "west"
  address_space       = ["10.0.0.0/16"]
}

resource "azurerm_mssql_database" "db1" {
  name      = "example-db"
  server_id = "server_db"
  collation = "sql_general"


data "azurerm_kubernetes_cluster" "aks" {
  name                = "prod-cluster"
  resource_group_name = "prod-group"
}
```

Provision Resources

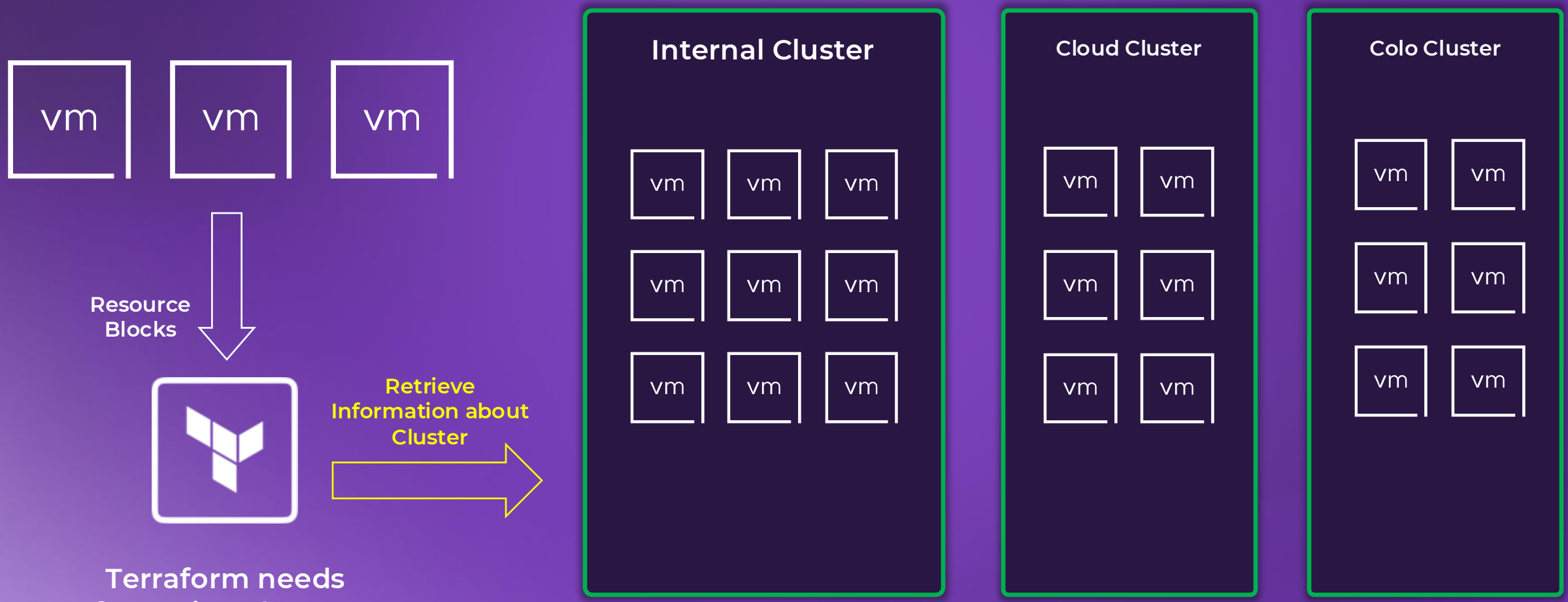
Retrieve Information

Use Data Block to Retrieve Information

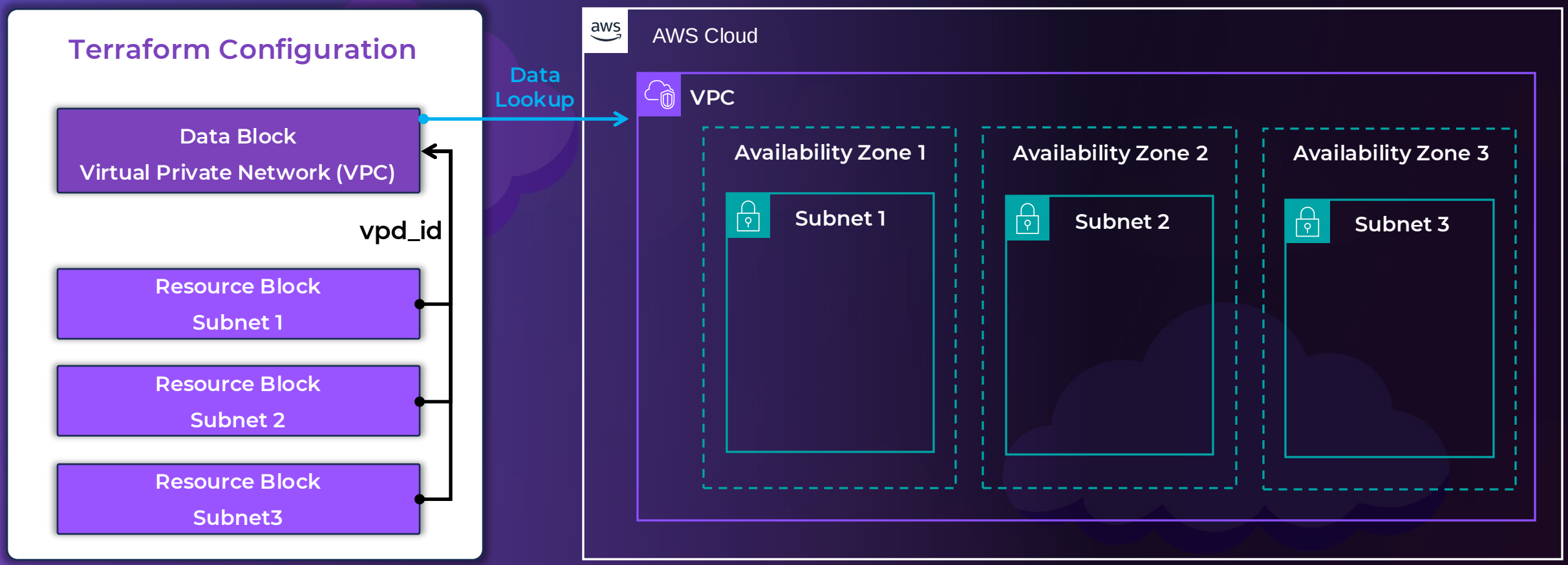
Existing Resources

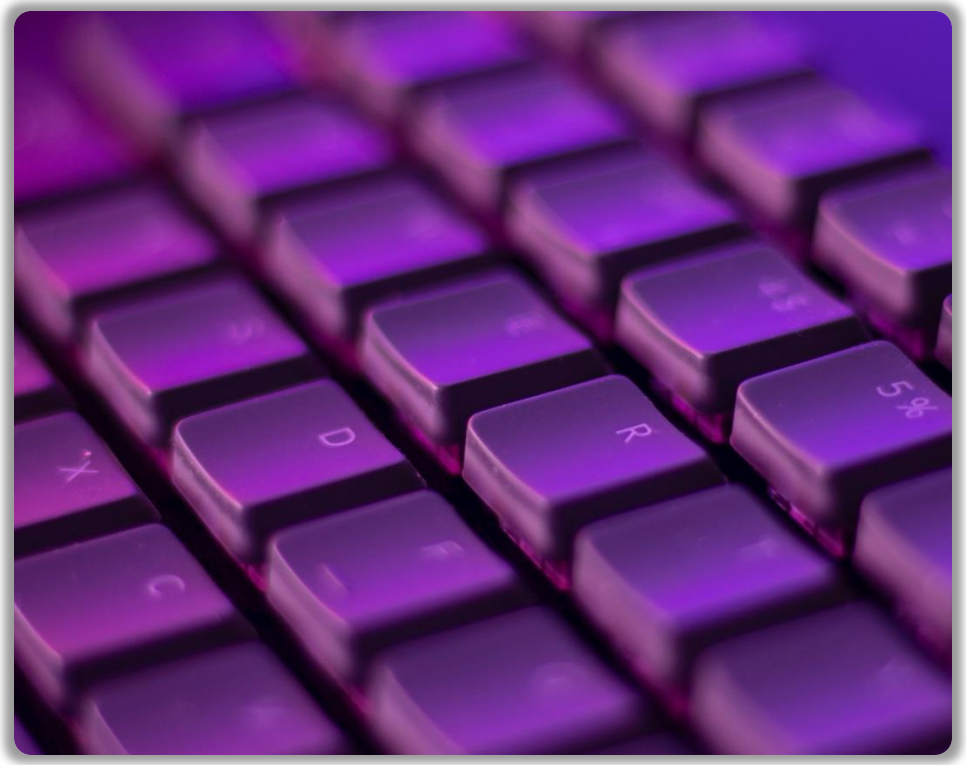


Existing Resources - Example



Existing Resources - Example





Data Block

A **`data`** block retrieves information about existing resources or external data without creating new infrastructure. It allows you to dynamically reference attributes, such as the ID of an existing subnet or the details of a Kubernetes cluster, making configurations flexible and reusable.

By using **`data`** blocks, you can integrate Terraform with resources managed outside of its configuration.

Data Block

Data blocks allow Terraform to access information about existing resources rather than creating new ones.

Access Existing Infrastructure

Retrieve the details of existing resources so they can be referenced in other Terraform blocks

Dependency Resolution

Data blocks help when one resource relies on the properties of another existing resource

Avoid Hardcoding Values

Data blocks prevent the need for hardcoded values by dynamically pulling in details, which makes configurations more adaptable

network.tf

```
data "aws_vpc" "prd" {
  filter {
    name   = "tag:Name"
    values = ["prd-vpc"]
  }
}

resource "aws_subnet" "pub" {
  vpc_id     = data.aws_vpc.prd.id
  cir_block = "10.0.6.0/24"
  ...
}

output "prd_vpc_subnet" {
  value = data.aws_vpc.prd.cidr_block
}
```

Breaking Down the Data Block

The given name used to reference the data block – user-defined string

The type of the resource to obtain data about - defined by provider

Query constraints defined by the data source to get the data you want

Get data about an Azure Resource Group named `dev-resource-group`

Get data about the default namespace for a Kubernetes cluster

main.tf

```
data "aws_vpc" "prd" {
  filter {
    name   = "tag:Name"
    values = ["prd-vpc"]
  }
}

data "azurerm_resource_group" "dev" {
  name = "dev-resource-group"
}

data "kubernetes_namespace" "app" {
  metadata {
    name = "customer-app"
  }
}
```

Referencing Data Blocks

Get information from a `data` block to use elsewhere within the configuration.

`data` blocks retrieve a wide range of information about the specified resource – called **`attributes`**.

Data block attributes are referenced just like resource attributes: `data.<type>.<name>.<attribute>`

- `data.aws_vpc.prd.arn`
- `data.aws_vpc.prd.cidr_block`
- `data.aws_vpc.prd.id`
- `data.aws_vpc.prd.owner_id`

network.tf

```
data "aws_vpc" "prd" {
  filter {
    name   = "tag:Name"
    values = ["prd-vpc"]
  }
}
```

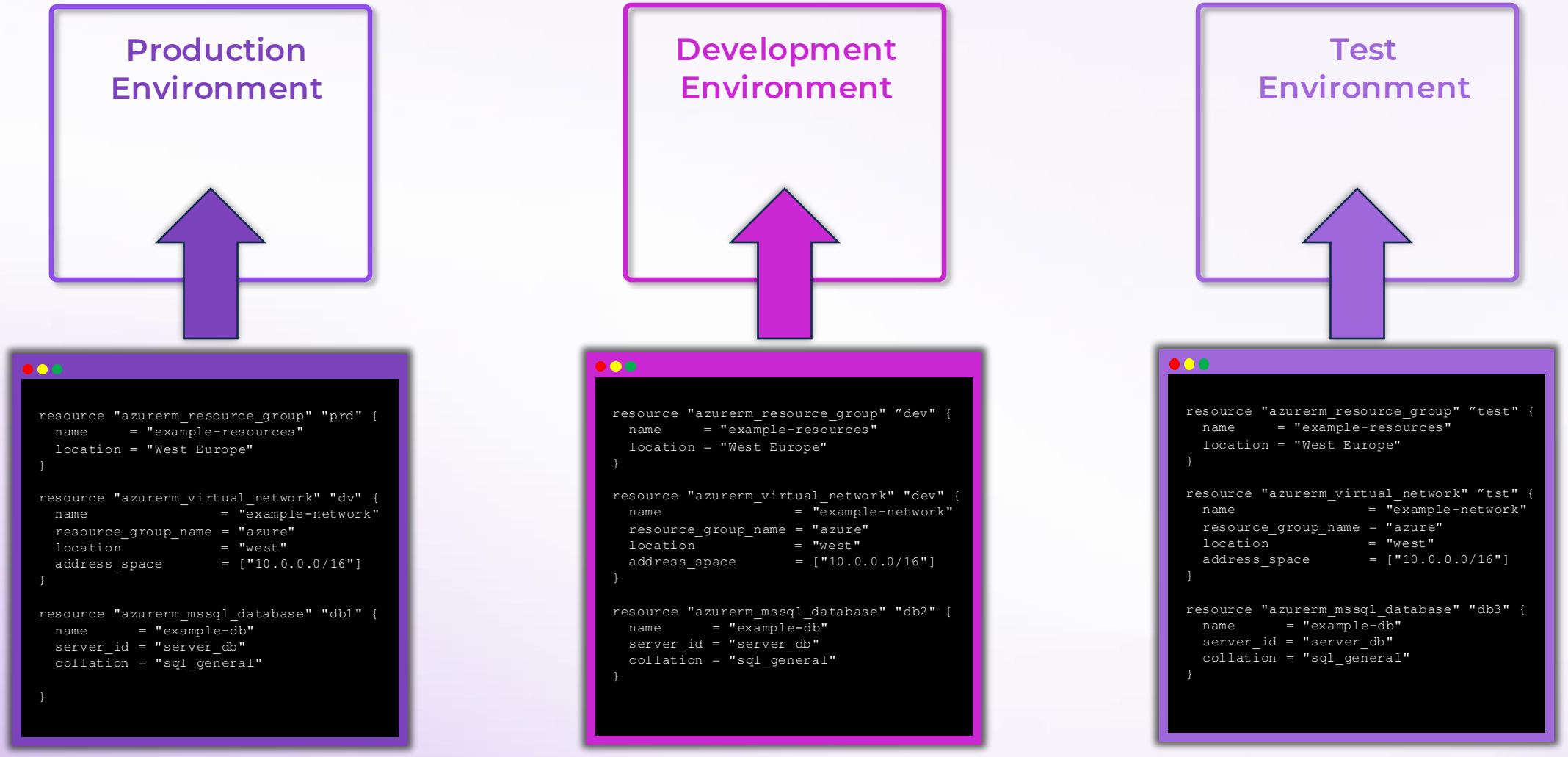
This data block gets all this data

```
{
  "arn": "arn:aws:ec2:us-west-2:xxx:vpc/vpc-0ab"
  "cidr_block": "10.0.0.0/16",
  "default": false,
  "dhcp_options_id": "dopt-0abcdef1234567890",
  "enable_dns_hostnames": true,
  "enable_dns_support": true,
  "id": "vpc-0abcd1234efgh5678",
  "instance_tenancy": "default",
  "ipv6_association_id": null,
  "ipv6_cidr_block": null,
  "main_route_table_id": "rtb-0abcdef1234567890",
  "owner_id": "123456789012",
  "tags": {
    "Name": "prd-vpc"
  }
}
```

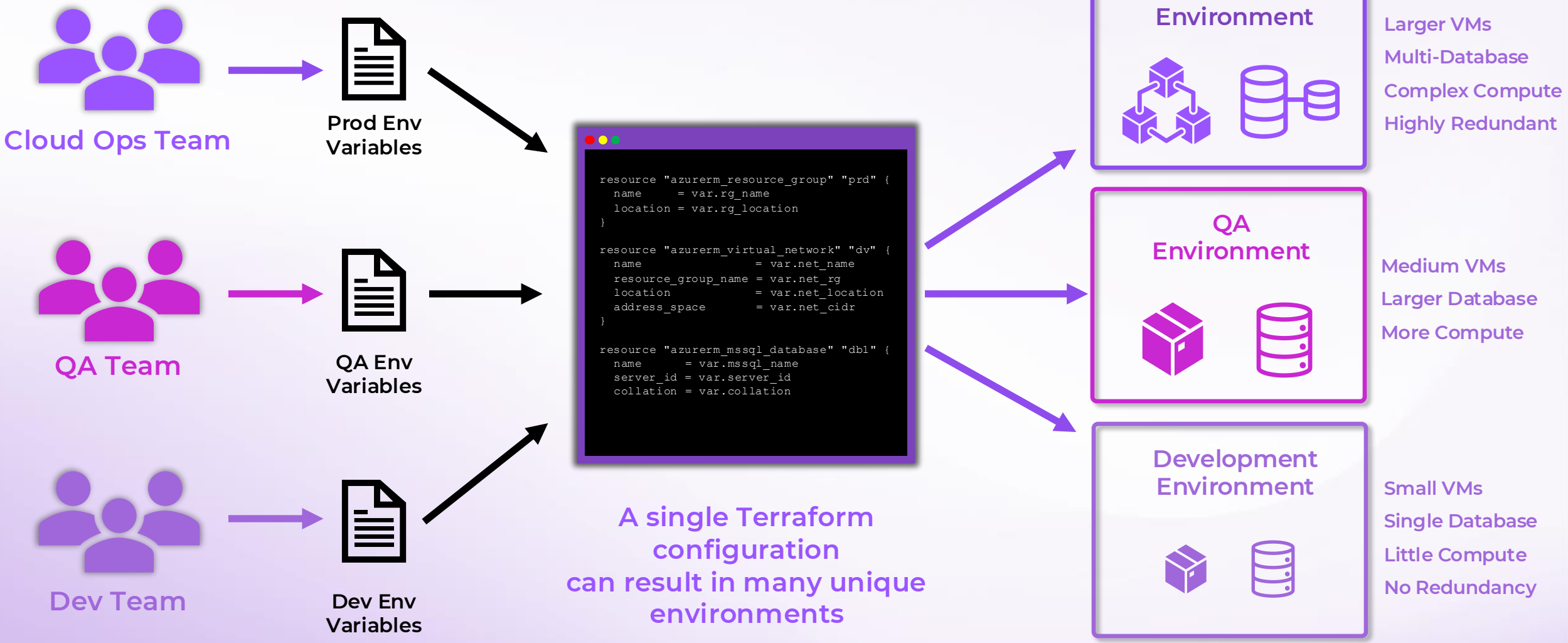
LECTURE

Variable Block

The Problem: Hardcoding Values in Terraform



The Solution: Using Variables in Terraform



Variable Block

A **`variable`** block is used to define inputs that can be reused and customized, making your configurations more flexible and dynamic. Variables allow you to parameterize values instead of hardcoding them directly into resource blocks.

This approach improves reusability, simplifies managing multiple environments, and helps maintain cleaner, more maintainable configurations.

Variable Block

Define an input that can be reused and customized to make configurations more flexible and dynamic

Dynamic Inputs for Dynamic Code

Variables allow configurations to adapt by using different values for different environments or scenarios.

Centralized Management of Values

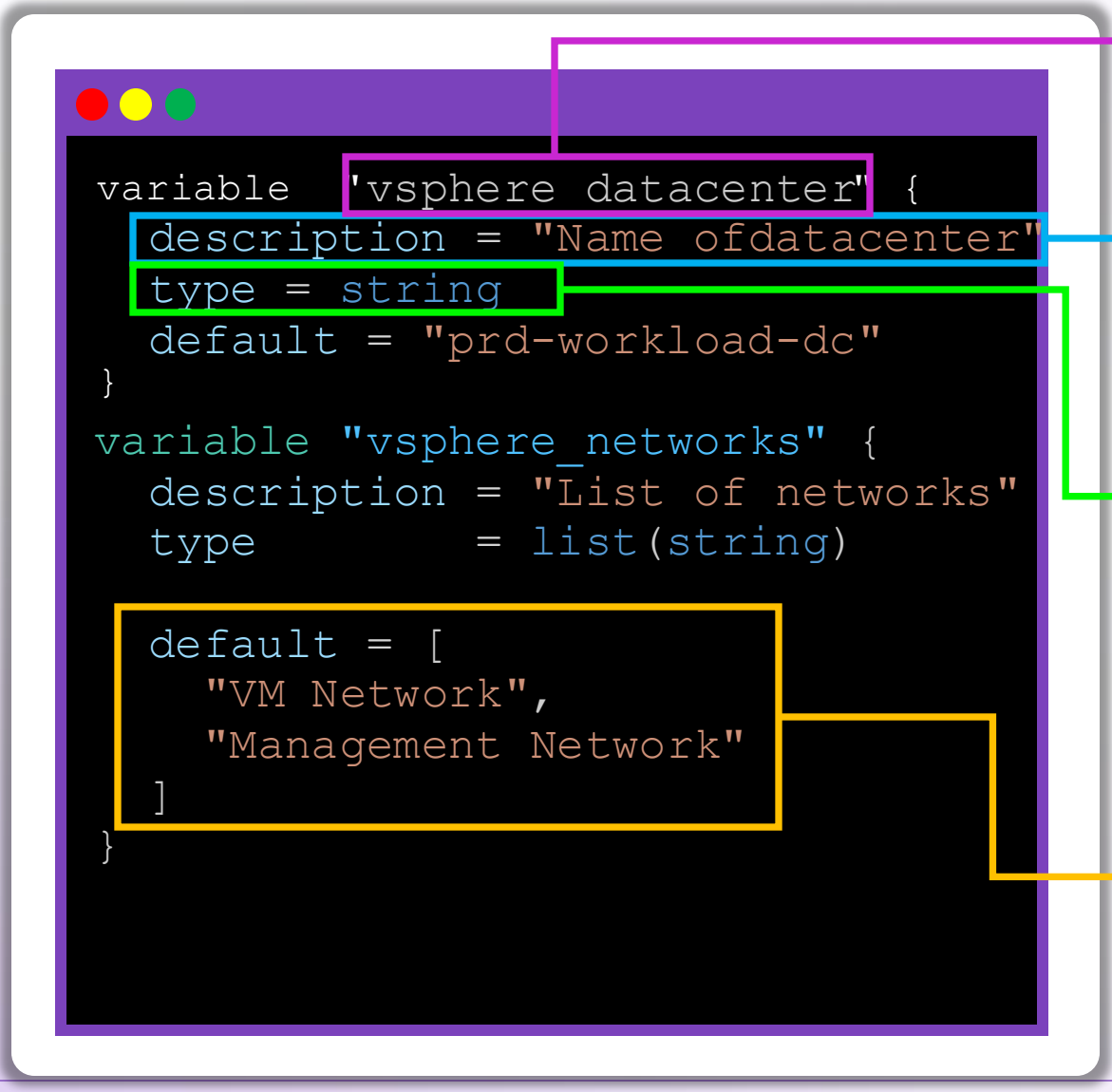
By defining variables in one place, you can update values easily without modifying multiple resource blocks.

Improved Reusability of Code

Variable blocks make Terraform configurations modular and reusable across projects and teams.

```
variable "vsphere_datacenter" {
  description = "Name of datacenter"
  type        = string
  default     = "prd-workload-dc"
}

variable "vsphere_networks" {
  description = "List of networks"
  type        = list(string)

  default     = [
    "VM Network",
    "Management Network"
  ]
}
```

Breaking Down the Variable Block



The name for the variable – a user-provided string

Friendly description of the variable and what it's used for...

Define what value types are accepted for the variable

The default value of the variable if not set anywhere else in Terraform

Types of Variables

Primitive Variable Types

ABC

String

a sequence of characters representing some text

```
variable "region" {
  type    = string
  default = "us-east-2"
}
```

9

Number

a numeric value

```
variable "num_of_vms" {
  type    = number
  default = 3
}
```

T/F

Bool

a boolean value – either True or False

```
variable "enable_ha" {
  type    = bool
  default = true
}
```

Types of Variables

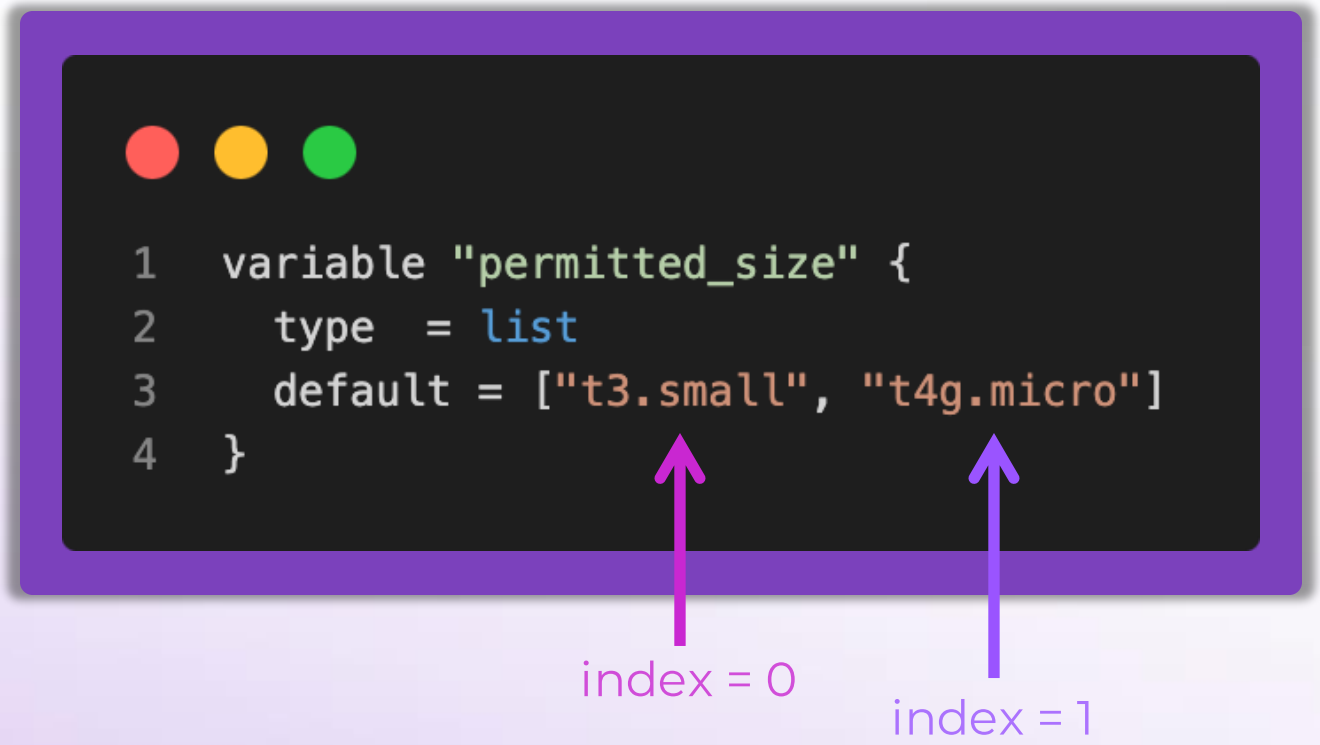
Complex Variable Types

List

A sequence of similar values

Presented by a pair of square brackets [ ] containing a comma-separated sequence of values

Note: lists start at 0



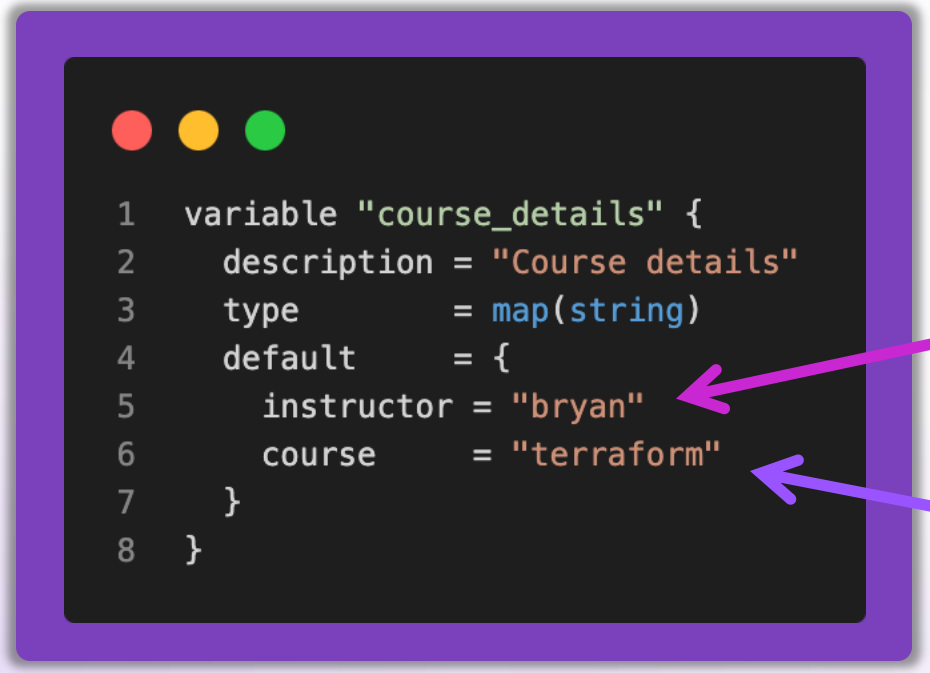
Types of Variables

Complex Variable Types

Map

A group of values identified by named labels

Presented by a pair of curly braces { } and contains a series of key=value pairs



Types of Variables

Complex Variable Types

Set

A collection of guaranteed unique values

Terraform does not support accessing elements of a set using an index because sets are unordered collections.

Note: Convert to a list to access

```
variable "pub_subnet_ids" {
  type    = set(string)
  default = ["subnet-12345", "subnet-67890"]
}
```

Resource Referencing

Reference a Variable within the Terraform configuration to use the value

Reference a Variable by using the **name** of the variable.

1. `var.vsphere_datacenter`
2. `var.vsphere_networks[0]`

```
variable "vsphere_datacenter" {
 description = "Name of datacenter"
 type        = string
 default     = "prd-workload-dc"
}

variable "vsphere_networks" {
 description = "List of networks"
 type        = list(string)

 default = [
  "VM Network",
  "Management Network"
 ]
}
```

1

2

Assigning Values to Variables

After variables are declared in your Terraform configuration, the values can be set in many different ways...

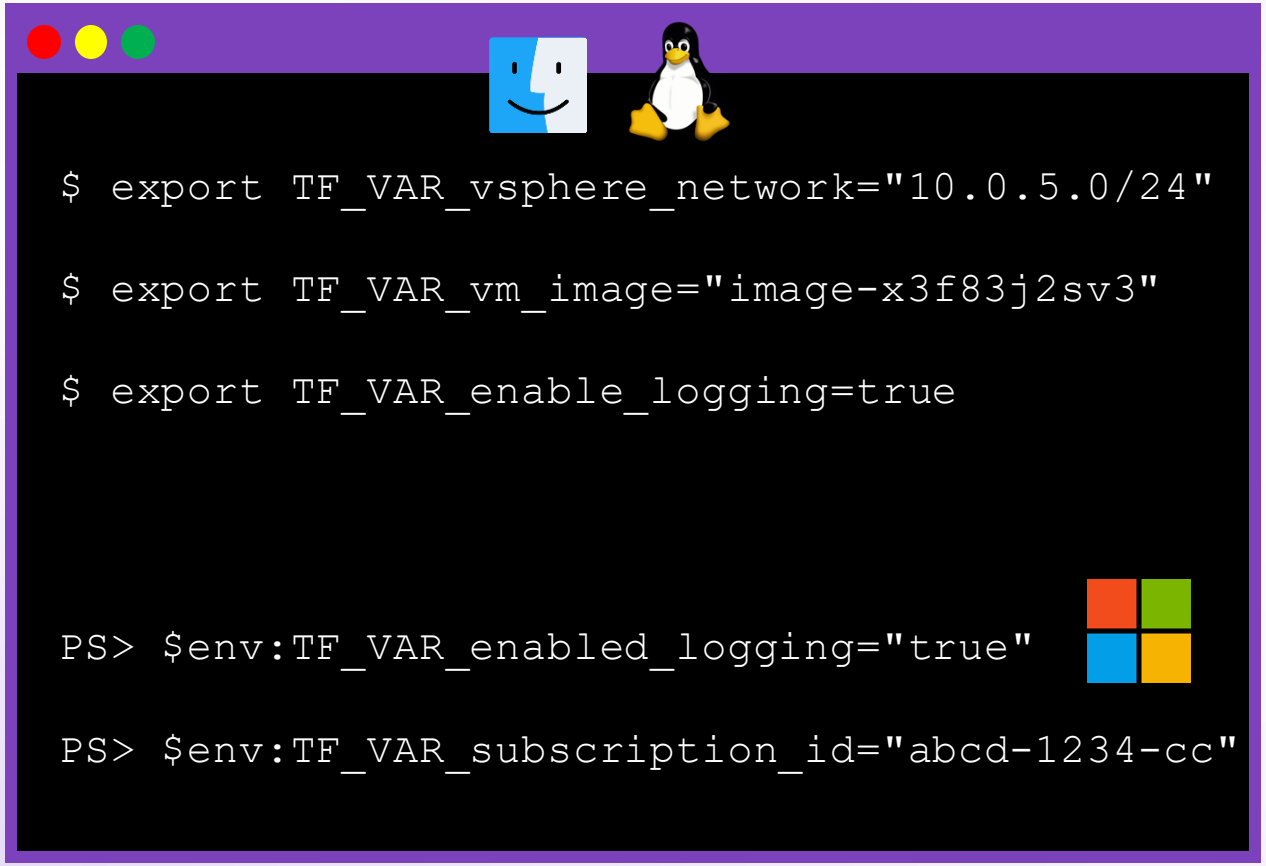
Variable Defaults

Set a value directly in the variable block declaration

```
1  variable "pub_subnet_ids" {
2    type    = set(string)
3    default = ["subnet-12345", "subnet-67890"]
4  }
```

Environment Variables

- Use an environment variable to set the value
- Must start with `TF_VAR_` followed by the variable name
- Useful for when you don't want to commit the value to a repo



tfvars File

- Use a `.tfvars` file to set variable values
- Terraform automatically loads a `.tfvars` file if they exist
- Set values using a `key=value` format using the variable names
- A `.tfvars` file is usually excluded from version control (.gitignore)

terraform.tfvars

```
# VM-level configurations
vsphere_network = "10.0.5.0/24"
vm_image        = "image-x3f83j2sv3"

# Application Configurations
enable_logging = true


# Cloud Account Configurations
subscription_id = "abcd-1234-cc"
```

Command Line Flags

- Pass variable values directly from the command line.
- Use `-var="key=value"` to set the value for a specific variable.
- Command-line `-var` values override both default values and values in `.tfvars` files.
- Useful for quick, temporary changes or testing without modifying files.

TERMINAL

```
$ terraform plan -var="enable_logging=true"

$ terraform apply -var="region=us-west-2" -var="vm_image=image-x3f83j2sv3"
```

Order of Precedence

What hierarchy does Terraform follow when resolving variable values from different sources?

Flexibility for Values

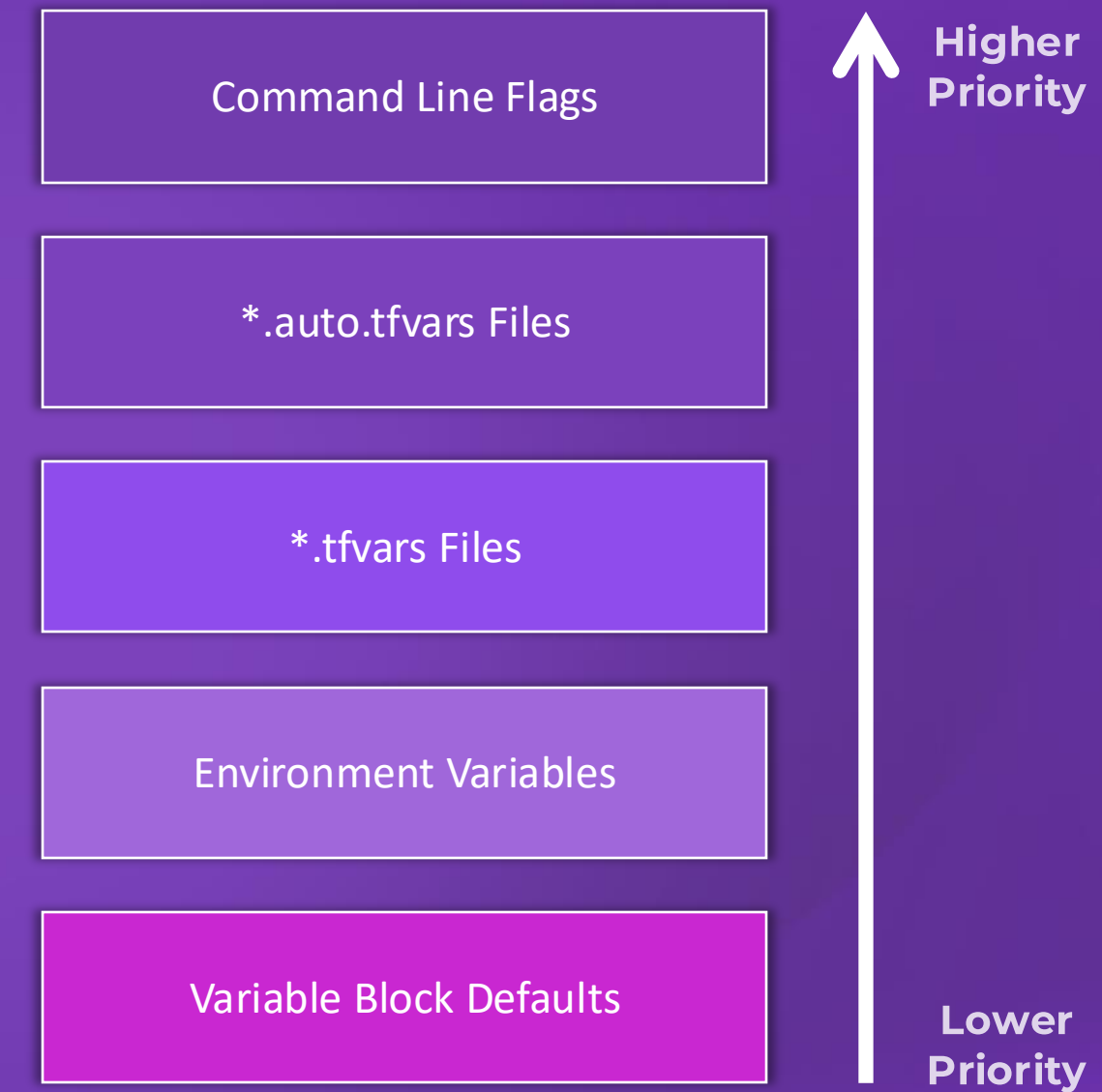
Understanding the order of precedence ensures the right value is applied

Clear Override Rules

Higher-priority sources override lower-priority defaults, preventing unexpected behavior.

Adaptability Across Environments

Precedence allows you to manage variables for different environments without changing configurations.



LECTURE

Output Block

Accessing New Resources

```
resource "azurerm_resource_group" "prd" {
  name     = "example-resources"
  location = "West Europe"
}

resource "azurerm_virtual_network" "dv" {
  name                = "example-network"
  resource_group_name = "azure"
  location            = "west"
  address_space       = ["10.0.0.0/16"]
}
```

Write Code

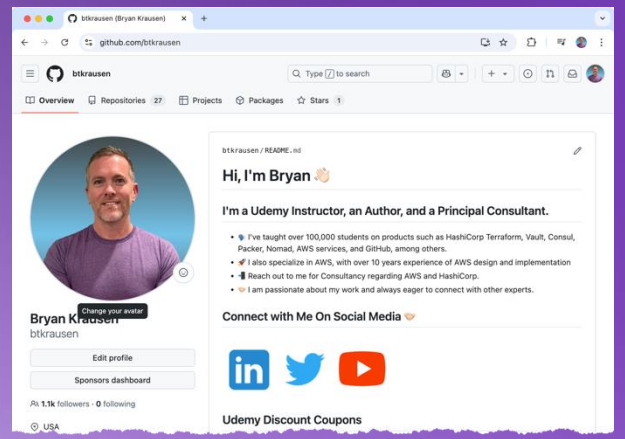
Provision Resources

New Resources

- Virtual Machine
- Kubernetes Service
- CosmosDB
- Storage Account
- Image Template
- Entra Domain

Output Information to Access Resources

https://btk.me/btk



Output Block

An **`output`** block in Terraform is used to display key information about your infrastructure after a deployment, such as IP addresses, DNS names, or resource IDs. Outputs make it easy to retrieve and share important details without manually looking them up in the cloud provider's console.

They're also useful for passing data between modules or automating workflows.

Output Block

An output block displays important information about your infrastructure, like resource IDs or URLs, after deployment.

Display Critical Information

Outputs make it easy to retrieve important details like IP addresses, DNS names, or resource IDs after deployment.

Enable Module Integration

Outputs can pass data between Terraform modules, simplifying workflows and making configs reusable.

Improve Workflow Efficiency

Outputs automate the retrieval of key values, reducing the need for manual lookups in provider consoles.

outputs.tf

```
# Output Instance IP Address
output "instance_public_ip" {
  description = "Public IP of Server"
  value       = aws_instance.web.public_ip
}

# Output DNS Name for Load Balancer
output "website_dns" {
  description = "Website DNS Record"
  value       = aws_elb.web_app.dns_name
}

# Output Friendly URL of Website
output "website_url" {
  value = "https://${aws_alb.web.dns_name}"
}
```

Breaking Down the Output Block

outputs.tf

```
# Output Instance IP Address
output "instance_public_ip" {
  description = "Public IP of Web server"
  value       = aws_instance.web.public_ip
}

# Output DNS Name for Load Balancer
output "website_dns" {
  description = "Website DNS Record"
  value       = aws_elb.web_app.dns_name
}

# Output Friendly URL of Website
output "website_url" {
  value = "https://${aws_alb.web.dns_name}"
}
```

The name of the output – user-defined string

The value of the output

Another output block – a more advanced value combining the value of a resource within a string

LECTURE

Terraform Block

Imagine This Real-World Scenario...

Your Workstation

```
resource "azurerm_resource_group" "prd" {
  name     = "example-resources"
  location = "West Europe"
}

resource "azurerm_virtual_network" "dv" {
  name                = "example-network"
  resource_group_name = "azure"
  location            = "west"
  address_space       = ["10.0.0.0/16"]
}

resource "azurerm_mssql_database" "db1" {
  name      = "example-db"
  server_id = "server_db"
  collation = "sql_general"

}
```

Commit Code

Git Repo

Pull Code

Colleague's Workstation

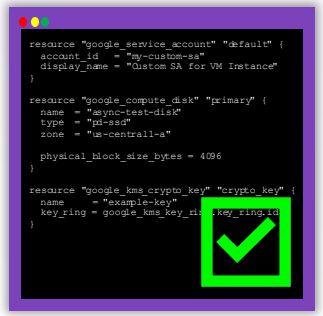
```
resource "azurerm_resource_group" "prd" {
  name     = "example-resources"
  location = "West Europe"
}

resource "azurerm_virtual_network" "dv" {
  name                = "example-network"
  resource_group_name = "azure"
  location            = "west"
  address_space       = ["10.0.0.0/16"]
}

resource "azurerm_mssql_database" "db1" {
  name      = "example-db"
  server_id = "server_db"
  collation = "sql_general"

}
```

Terraform Version 1.10

Terraform Version 1.11

The Risk of Mismatched Provider Versions

CloudOps Team



Commit Code

Git Repo

Pull Code

Data Research Team

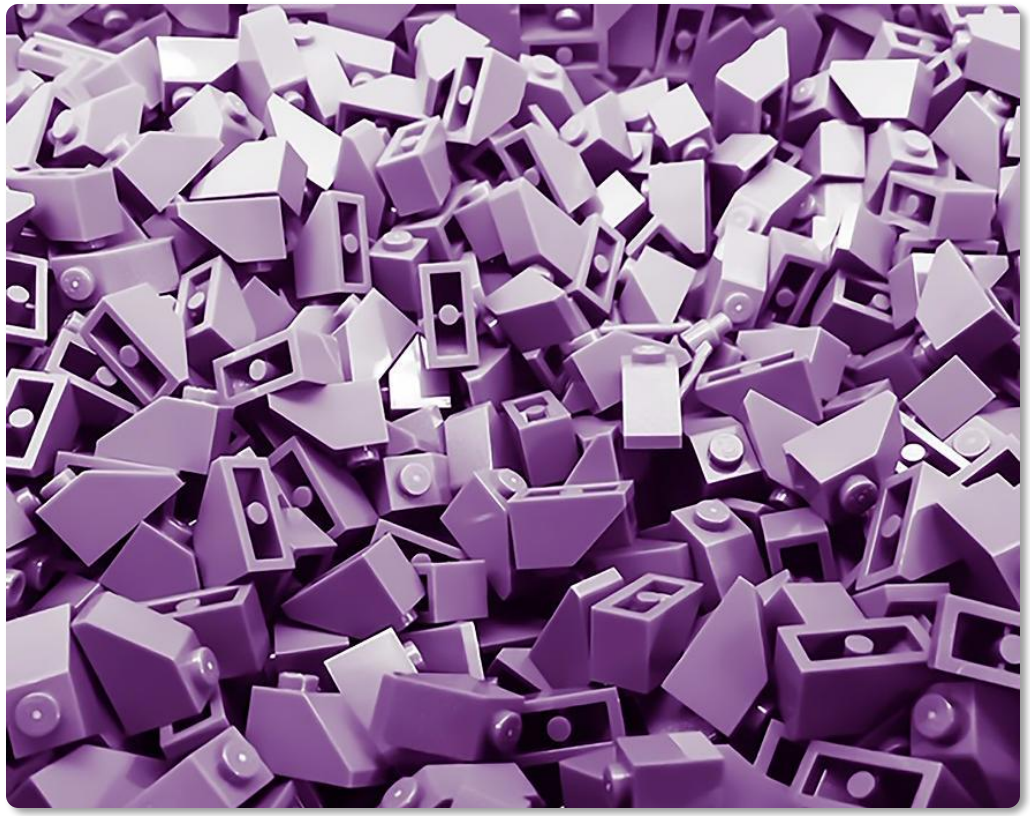
Adds Newly Released Service

Downloads Latest GCP Provider Version

Google Cloud Platform

GCP Provisioned Resources

Google Cloud Platform



Terraform Block

The `terraform` block sets global configurations for your Terraform project, such as required provider versions, Terraform version constraints, and backend settings for state management. It ensures consistency across teams and environments by enforcing specific versions and configurations.

Using the `terraform` block helps avoid errors and provides a solid foundation for managing infrastructure effectively.

Terraform Block

The `terraform` block ensures your Terraform setup is consistent, and scalable across teams and environments.

Version Control

Specifies the required Terraform and provider versions to avoid compatibility issues

Backend Configuration

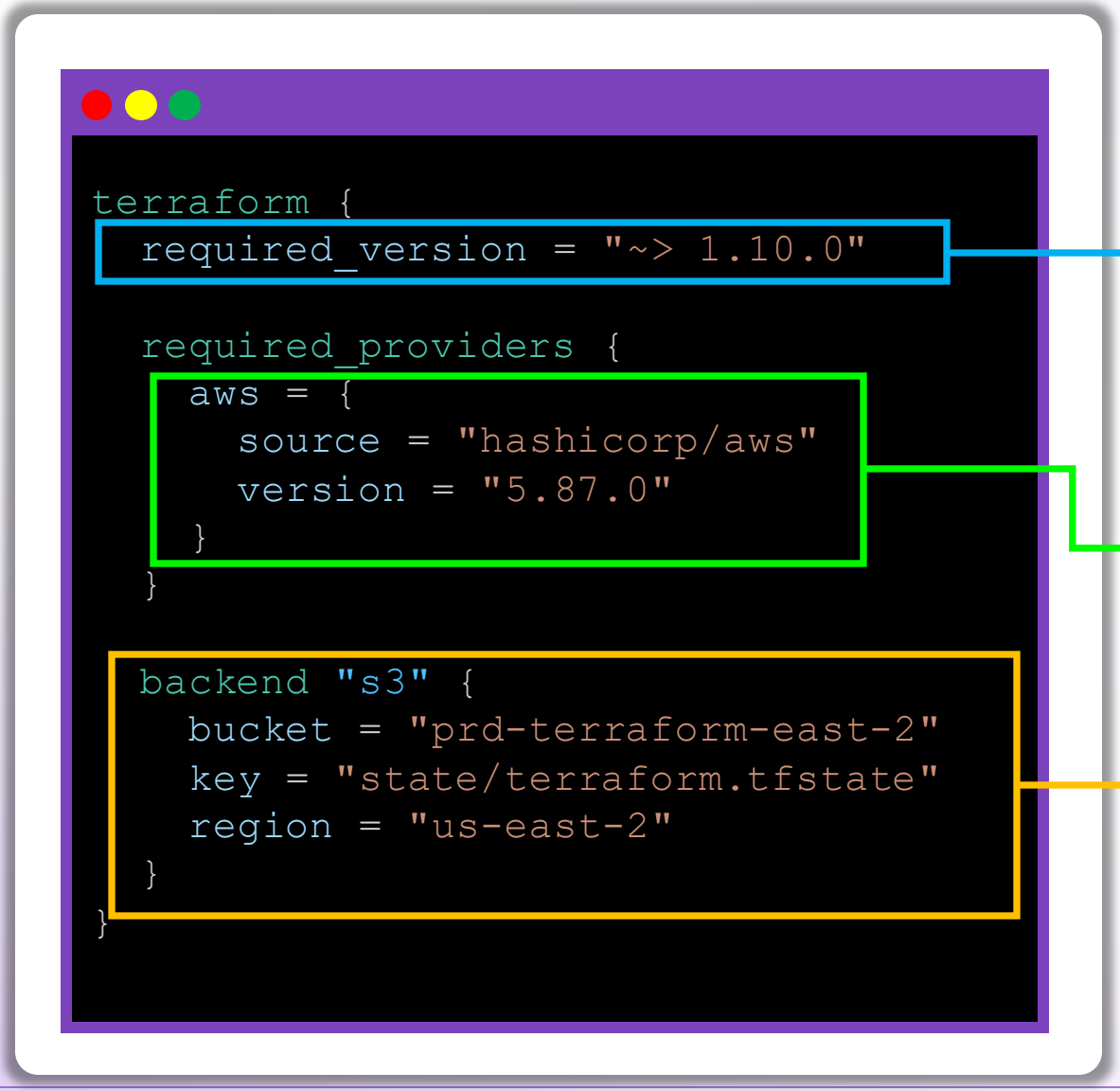
Configures where Terraform state is stored to ensure secure and reliable state management

Provider Requirements

Defines the providers and the versions needed for your infrastructure

```
terraform {
  required_version = "~> 1.10.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }

  backend "s3" {
    bucket = "prd-terraform-east-2"
    key    = "state/terraform.tfstate"
    region = "us-east-2"
  }
}
```

Breaking Down the Terraform Block



Ensures Terraform version 1.10.x is used to run this configuration

Specify a required provider and require version 5.87.0

The backend configuration of where you want to store Terraform state

Version Constraints

```
terraform {
  required_version = "1.9.8"

  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.77.0"
    }
  }
}
```

- Only Terraform v1.9.8 can be used to run this configuration
- Installs/uses AWS provider version 5.77.0

```
terraform {
  required_version = ">= 1.9.8"

required_providers {
    aws = {
      source = "hashicorp/aws"
      version = ">= 5.77.0"
    }
  }
}
```

- Terraform v1.9.8 or greater can be used to run this configuration
- Installs/uses AWS provider version greater or equal to 5.77.0

```
terraform {
  required_version = "~> 1.11.0"

required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 5.87.0"
    }
  }
}
```

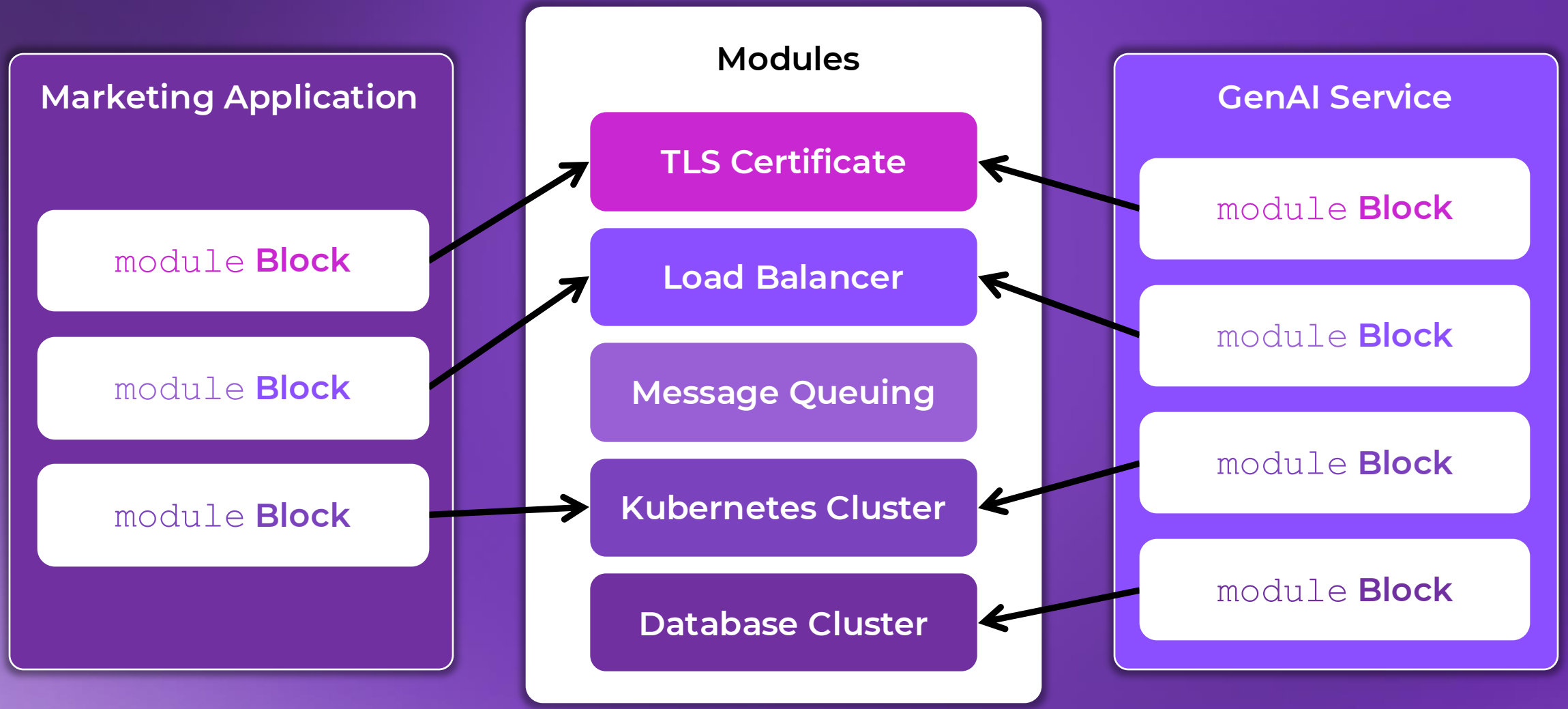
- Terraform v1.11.x or greater can be used to run this configuration
- Installs/uses AWS provider version greater or equal to 5.87.x

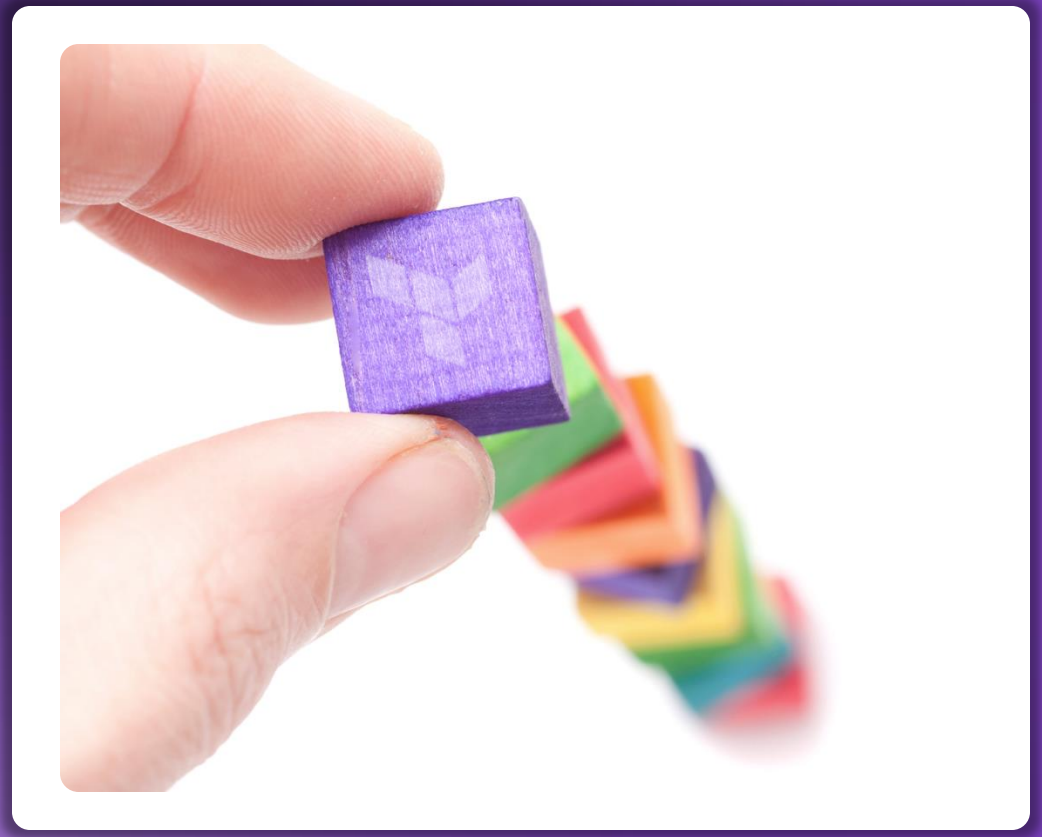
~ allows only the right-most version component to increment

LECTURE

Module Block

Terraform Building Blocks





Module Block

The **`module`** block allows you to reuse and organize Terraform configurations by calling pre-defined modules, which will simplify infrastructure management. Modules allow you to group resources into reusable components, such as a network or database module.

By passing variables to modules and using outputs, you can integrate them into your configuration for more dynamic and flexible deployments.

Module Block

Modules help you create maintainable and standardized infrastructure by packaging commonly used resources together.

Code Reusability

Create standardized infrastructure patterns that can be used across multiple projects and teams.

Simplified Management

Break complex infrastructure into smaller, manageable components that are easier to maintain and update.

Consistent Deployment

Ensure infrastructure consistency by using tested, version-controlled module configurations across environments.

main.tf

```
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"
  version = "4.0.1"

  name = var.vpc_name
  cidr = var.vpc_cidr_block

  azs = ["us-west-2a", "us-west-2b"]
  private_subnets = ["10.0.1.0/24"]
  public_subnets  = ["10.0.101.0/24"]

  enable_nat_gateway = true
  single_nat_gateway = true
}
```

Breaking Down the Module Block

main.tf

```
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"
  version = "4.0.1"

  name = var.vpc_name
  cidr = var.vpc_cidr_block

  azs = ["us-west-2a", "us-west-2b"]
  private_subnets = ["10.0.1.0/24"]
  public_subnets  = ["10.0.101.0/24"]

  enable_nat_gateway = true
  single_nat_gateway = true
}
```

Where the module is located to download

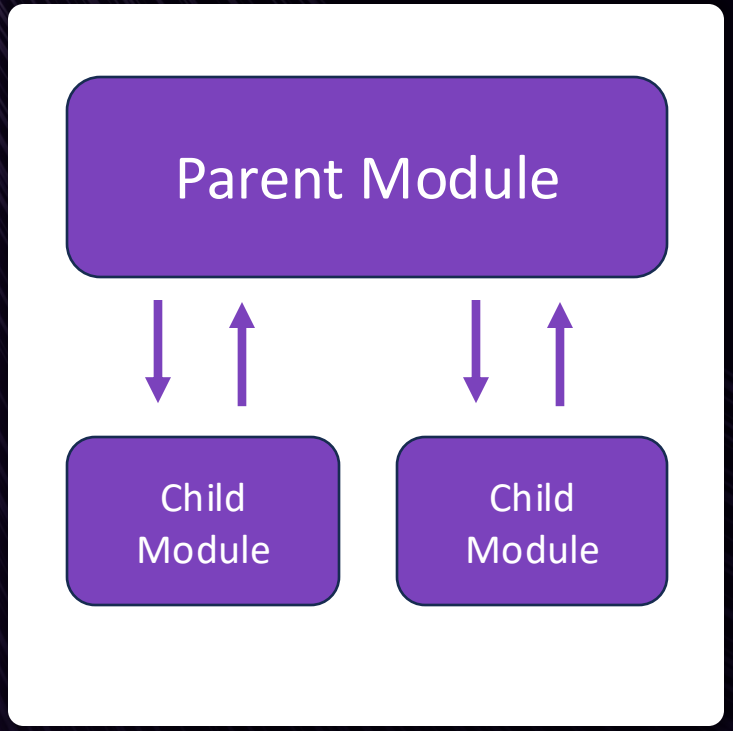
The version of the module to use

Local variables/values to pass into the module

Module Block

Parent Module vs. Child Module

- A parent (calling) module references and configures other modules, while a child module is the reusable infrastructure component being called.
- Modules can be retrieved from a registry, a repo, or stored locally.



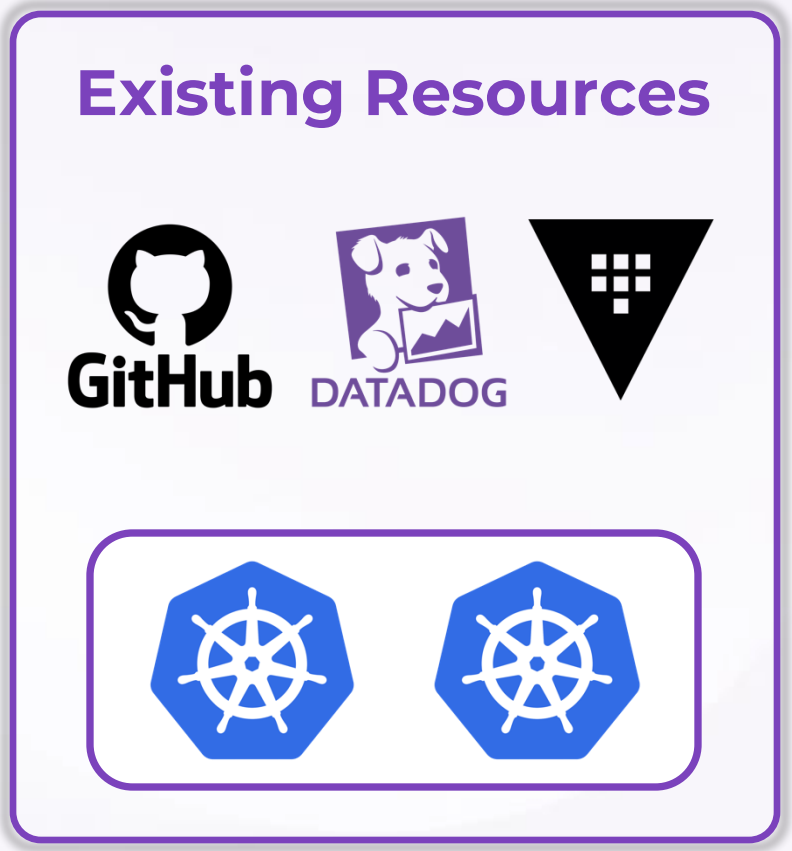
LECTURE

Import Block

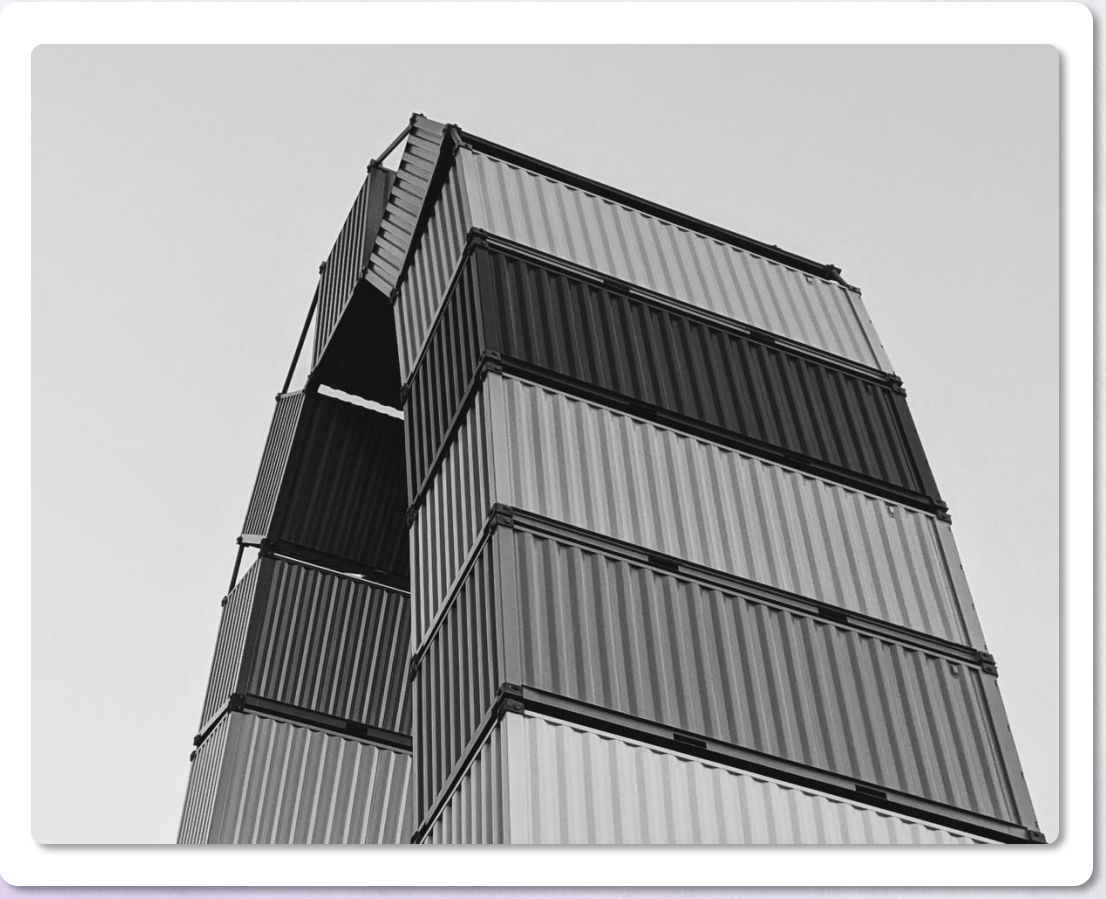
Manage all the Things with Terraform…

```
resource "aws_instance" "web" {
  ami           = "ami-012345"
  instance_type = "t2.micro"

  key_name   = "prd-web-key"
  subnet_id = "subnet_12345abc"

  tags = {
    Name = "prd-web-svr_01"
  }
}

resource "aws_instance" "db" {
  instance_type = "m5.4xlarge"
  ami           = "ami-0c55b159"
}
```

After learning Terraform, you want to start provisioning all resources using Infrastucture as Code



…but what about these resources already running in the cloud/on-premises?



Import Block

The `import` block in Terraform lets you bring existing infrastructure under Terraform management by mapping real-world resources to your configuration. This allows Terraform to track and manage them without manual setup.

It simplifies migration, reduces effort, and ensures infrastructure is versioned and controlled.

Import Block

Define an input that can be reused and customized to make configurations more flexible and dynamic

Map Existing Resources

Links real-world infrastructure to Terraform without changing it

Simplifies Migration

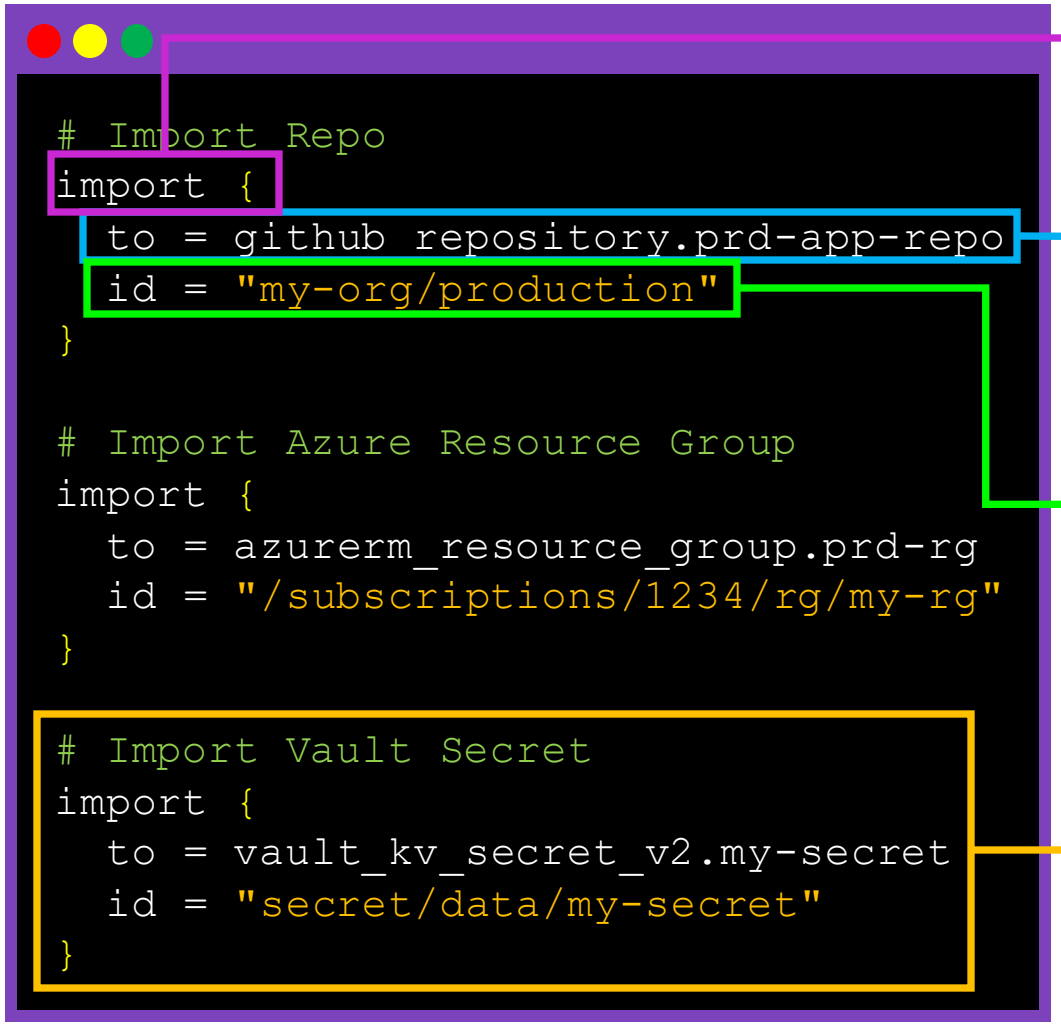
Reduces manual setup when adopting Terraform

Enables State Management

Bring unmanaged resources under Terraform control

```
# Import Repo
import {
  to = github_repository.prd-app-repo
  id = "my-org/production"
}

# Import Azure Resource Group
import {
  to = azurerm_resource_group.prd-rg
  id = "/subscriptions/1234/rg/my-rg"
}

# Import Vault Secret
import {
  to = vault_kv_secret_v2.my-secret
  id = "secret/data/my-secret"
}
```

Breaking Down the Import Block



The import block & curly braces

What resource block in Terrform will this be mapped to?

What is the actual resource that you want to import? A VM? A repo? A network?

Another example of importing a resource from Vault

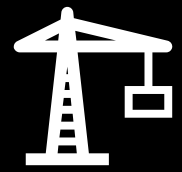
Import Block - Requirements

- Before importing a resource, you need to create the resource block that will manage the imported resource moving forward.
- You can write it manually or use `terraform plan -generate-config-out=PATH`

```
# Import Repo
import {
  to = github_repository.prd-app
  id = "my-org/production"
}

# Import Vault Secret
import {
  to = vault_kv_secret_v2.secret
  id = "secret/data/my-secret"
}
```

```
resource "github_repository" "prd-app" {
  name        = "production"
  description = "Production Codebase"
}


resource "vault_kv_secret_v2" "secret" {
  mount     = "secret"
  name      = my-secret
  data_json = jsoncode ({admin=pass"})
}
```

SUMMARY



Blocks are the fundamental building blocks of Terraform configurations, each serving a specific purpose in IaC

Output blocks enable visibility into deployment results, crucial for monitoring and debugging

Provider and resource blocks form the core of infrastructure deployment, connecting to platforms and defining what to create

Module and import blocks promote code reuse and easier management of existing infrastructure

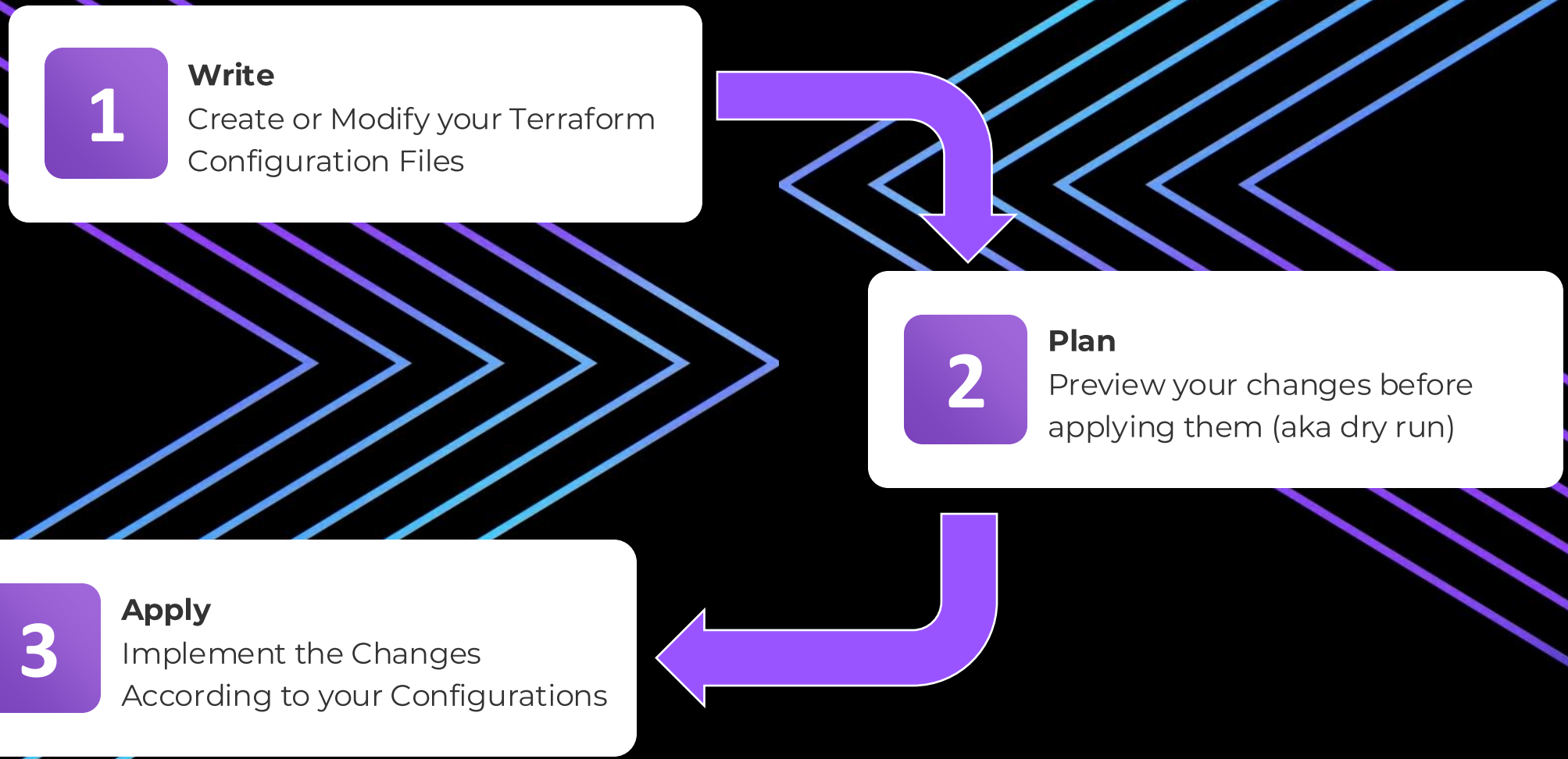
Data and variable blocks enhance flexibility by allowing dynamic value injection and existing resource queries

Together, these blocks create a complete framework for infrastructure lifecycle management

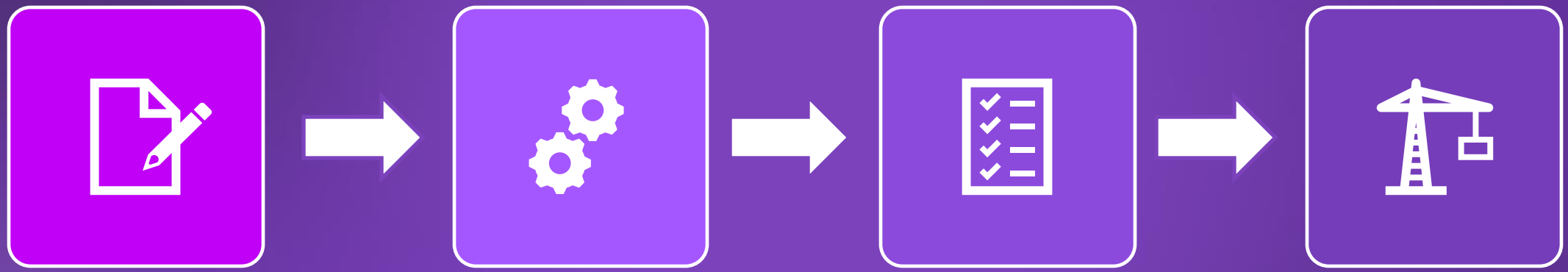
LECTURE

Introduction to the Terraform Workflow

The Terraform Workflow (Official)



Terraform Workflow



Write Configuration (HCL)

`Initialize`

`Plan`

`Apply`

Developing Terraform Configuration

What Does "Develop Configuration" Mean?

The Write stage involves creating one or more configuration files that define your desired state – the end result you want for your infrastructure. Terraform configurations are iterative, meaning you can build and refine your configuration incrementally.

Defining Desired State

You describe the infrastructure components (e.g., VMs, Networks) and the properties for each. Terraform ensures that the "real-world" infrastructure matches this state

`terraform init`

`terraform init` **is the first command you run in a new Terraform project to set up your working directory**

What Does It Do?

It prepares the environment to run other Terraform commands. Downloads the required providers (e.g., AWS, Azure, GCP) and modules used in the Terraform configuration.

Backend Management

`terraform init` also configures the backend for storing Terraform state

When Do You Use It?

At the start of a new project or any time you add, remove, or upgrade providers/modules or change the backend settings in your configuration.

terraform init

TERMINAL

```
$ terraform init
Initializing the backend...
Initializing provider plugins...
- Finding latest version of hashicorp/vsphere...
- Finding latest version of hashicorp/random...
- Finding latest version of hashicorp/vault...
- Installing hashicorp/vault v4.5.0...
- Installed hashicorp/vault v4.5.0 (signed by HashiCorp)
- Installing hashicorp/vsphere v2.10.0...
- Installed hashicorp/vsphere v2.10.0 (signed by HashiCorp)
- Installing hashicorp/random v3.6.3...
- Installed hashicorp/random v3.6.3 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

`terraform plan`

`terraform plan` generates an execution plan showing the changes that Terraform will make to the 'real-world' resources in order to match the desired state configuration

What Does It Do?

Provides you with an output of the changes that will happen if you apply the changes
It shows you the resources to be created (+), updated (~), or destroyed (-).

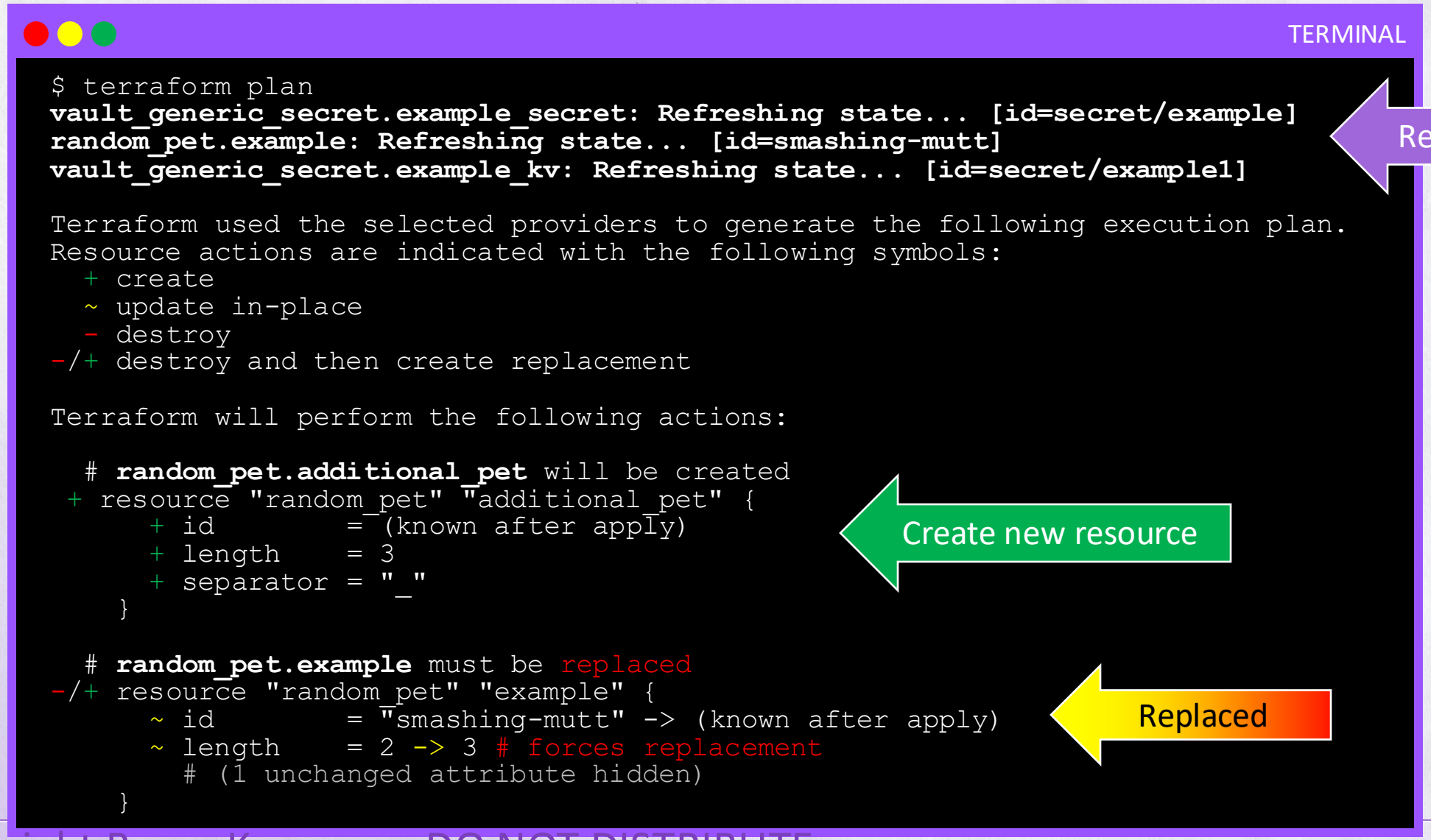
Perform a Dry-Run of Your Changes

`terraform plan` gives you the ability to perform a 'dry-run' without impacting 'real-world' resources. It helps avoid unintended changes by showing exactly what Terraform will do

Why a Plan is Important

It is crucial for reviewing changes in production environments to prevent downtime out errors.changes,uld be run before applying any changes but it is **NOT** required for a `terraform apply`

terraform plan

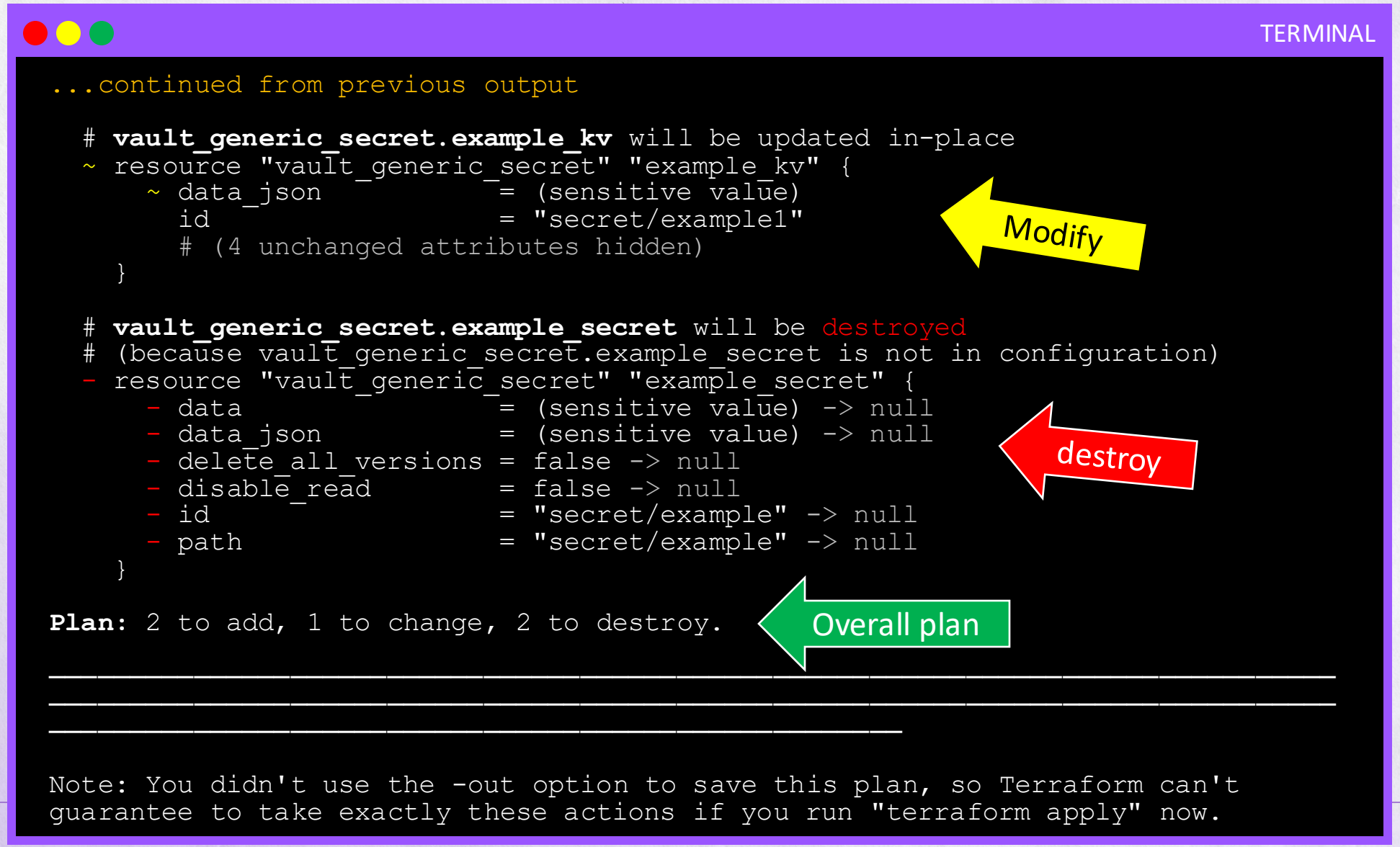


Refreshes State

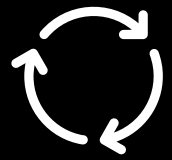
Create new resource

Replaced

terraform plan



SUMMARY



Terraform workflow begins with `terraform init`, which sets up the project and downloads necessary providers and modules.

`terraform plan` generates an execution plan, allowing you to preview changes before applying them

`terraform apply` is then used to make the planned changes to your infrastructure.

By default, Terraform will ask for confirmation before making any changes to real-world infrastructure

Each step in the workflow is crucial for managing infrastructure predictably and efficiently.

Following this workflow ensures consistency and control over infrastructure changes.

LECTURE

Using the Terraform CLI

What is the Terraform CLI?

The Terraform CLI is the primary way to interact with Terraform and execute the workflow steps

Unified Workflow

The CLI ensures consistency and repeatability across your workflow. – regardless of what platform that you're using Terraform

Command Interface

A command-line tool to manage infrastructure through commands like terraform init, plan, and apply

Consistent Pattern

Follows a consistent pattern:

```
terraform <command> [options]

tofu <command> [options]
```

Getting to Know the Terraform CLI

Terraform commands follow this structure:

→ `terraform <subcommand> [options or flags]`

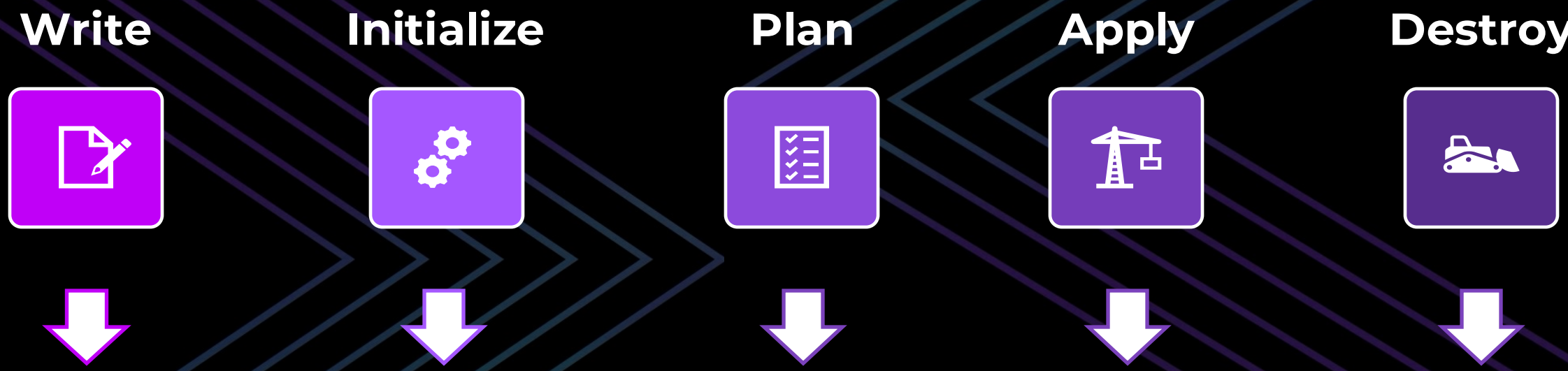
```
terraform   plan   -out=planfile
```

Base command to invoke the CLI

Subcommand

Optional Flag

Mapping the Workflow to Terraform Commands



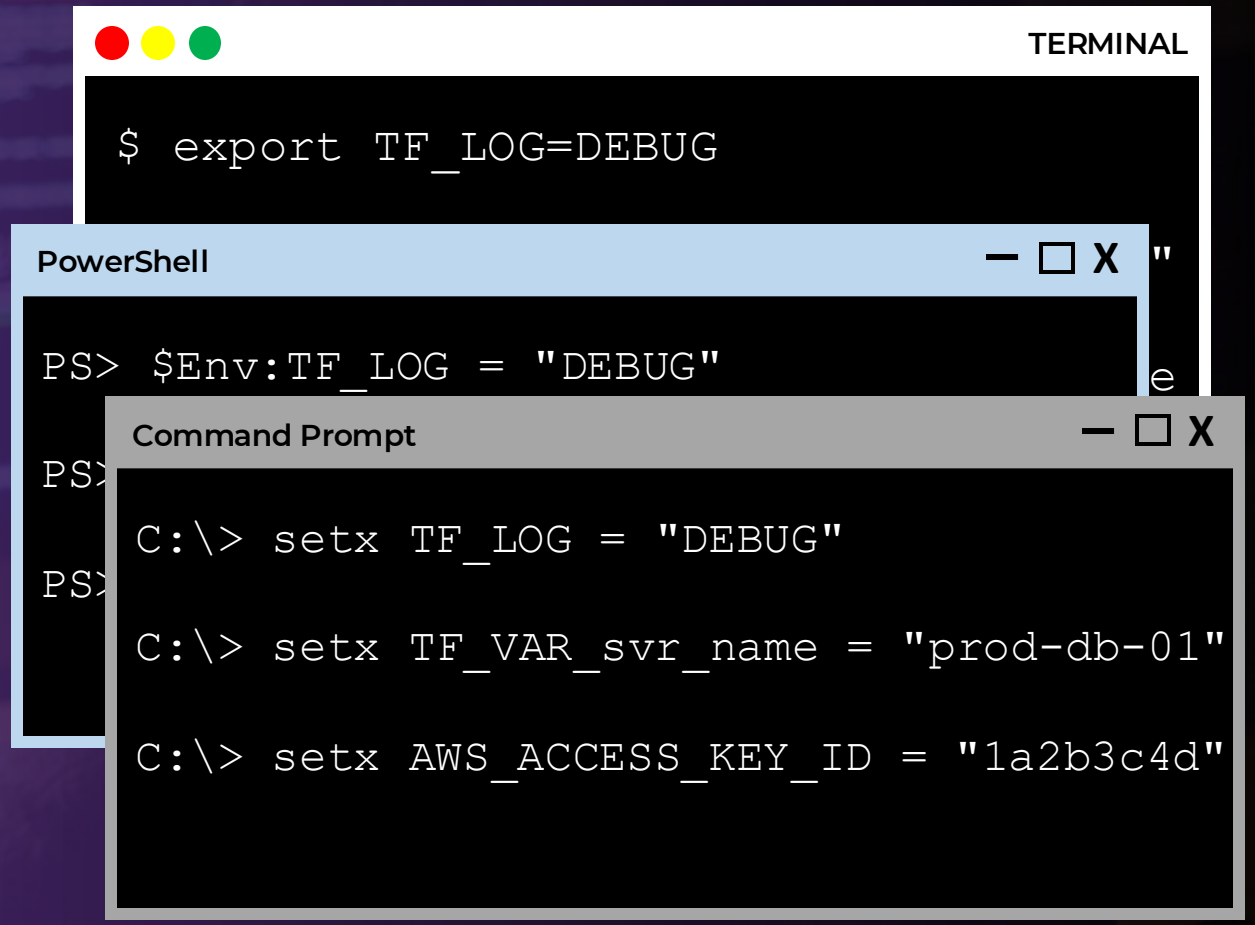
Terraform Subcommands

- [x] apply
- [ ] console
- [x] destroy
- [x] fmt
- [ ] force-unlock
- [ ] get
- [ ] graph
- [x] import
- [x] init
- [ ] login
- [ ] logout
- [ ] modules
- [x] output
- [x] plan
- [ ] providers
- [x] show
- [x] state
- [ ] test
- [x] validate
- [x] version
- [ ] workspace

*not an exhaustive list

Using Environment Variables with the CLI

- Environment variables provide a way to pass configuration settings and credentials to Terraform without hardcoding them into your files.
- They increase security by keeping sensitive data like API keys out of `.tf` files.



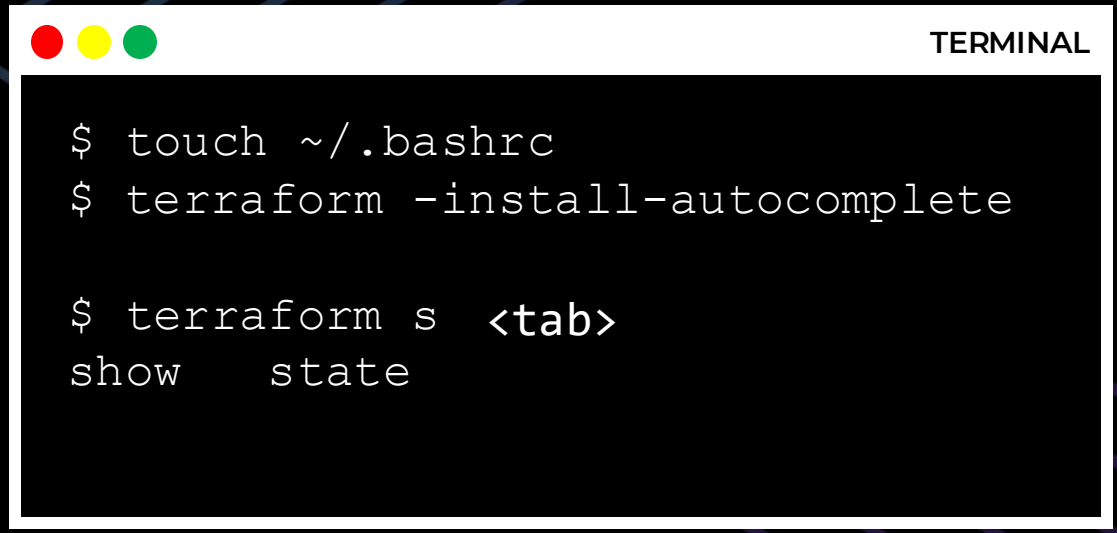
Making the Most of the Terraform CLI

Auto-Complete

Tab completion helps you quickly fill in subcommands, flags, and file paths by pressing the **Tab** key.

It reduces typing errors and ensures you use valid options for Terraform commands.

Available for most shell environments (like Bash or Zsh) after enabling it with terraform -install-autocomplete.



Making the Most of the Terraform CLI

Using the `help` feature

TERMINAL

```
$ terraform --help
Usage: terraform [global options] <subcommand> [args]

The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.

Main commands:
  init          Prepare your working directory for other commands
  validate      Check whether the configuration is valid
  plan          Show changes required by the current configuration
  apply         Create or update infrastructure
  destroy       Destroy previously-created infrastructure
...
```

Making the Most of the Terraform CLI

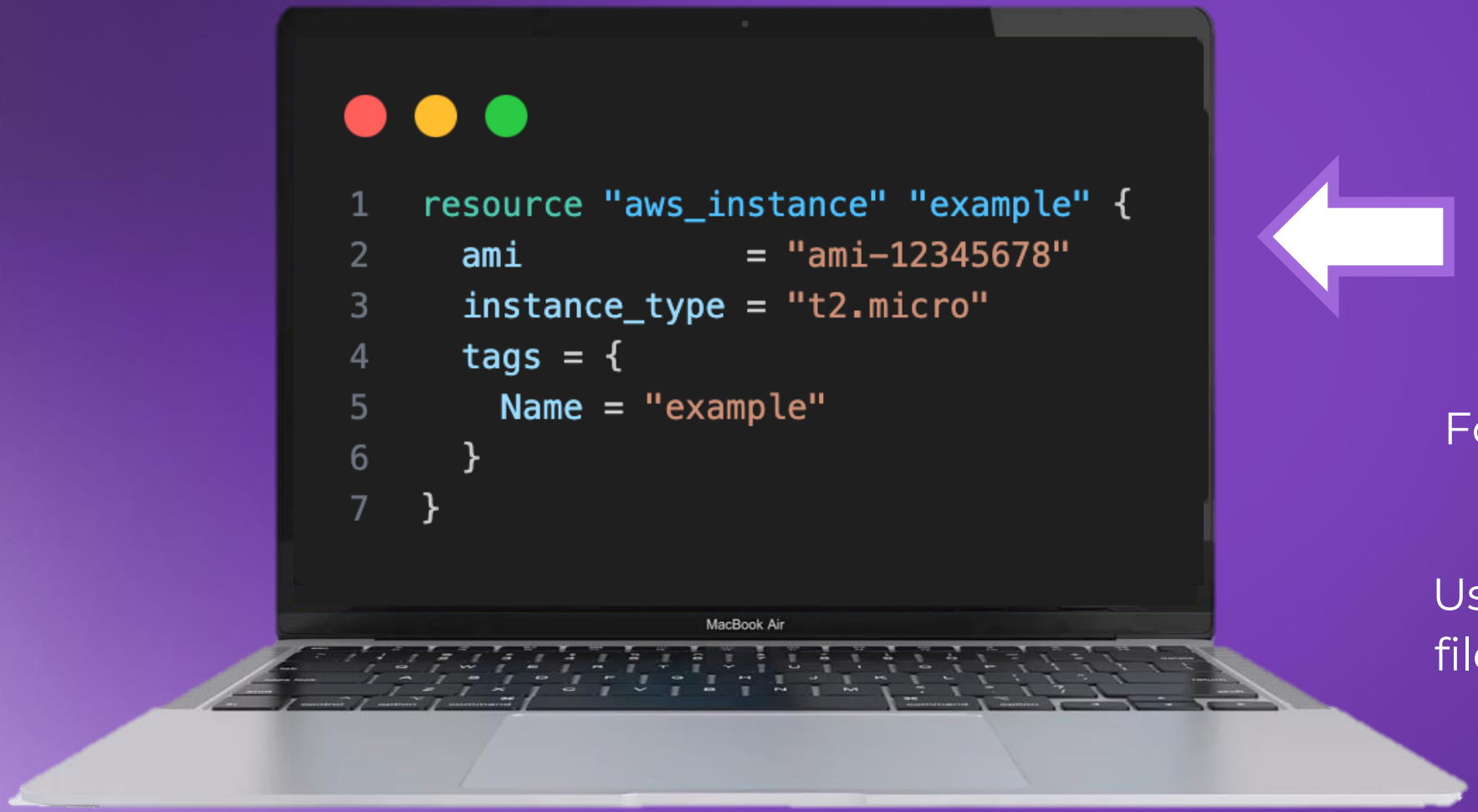
Using the **`help`** feature

TERMINAL

```
$ terraform plan --help
Usage: terraform [global options] plan [options]

  Generates a speculative execution plan, showing what actions Terraform
  would take to apply the current configuration. This command will not
  actually perform the planned actions.

  You can optionally save the plan to a file, which you can then pass to
  the "apply" command to perform exactly the actions described in the plan.

Plan Customization Options:

  The following options customize how Terraform will produce its plan. You
  can also use these options when you run "terraform apply" without passing
  it a saved plan, in order to plan and apply in a single command.

  -destroy           Select the "destroy" planning mode, which creates a plan
                     to destroy all objects currently managed by this
```

Making the Most of the Terraform CLI

Easily Format Your Code To a Canonical Format and Style

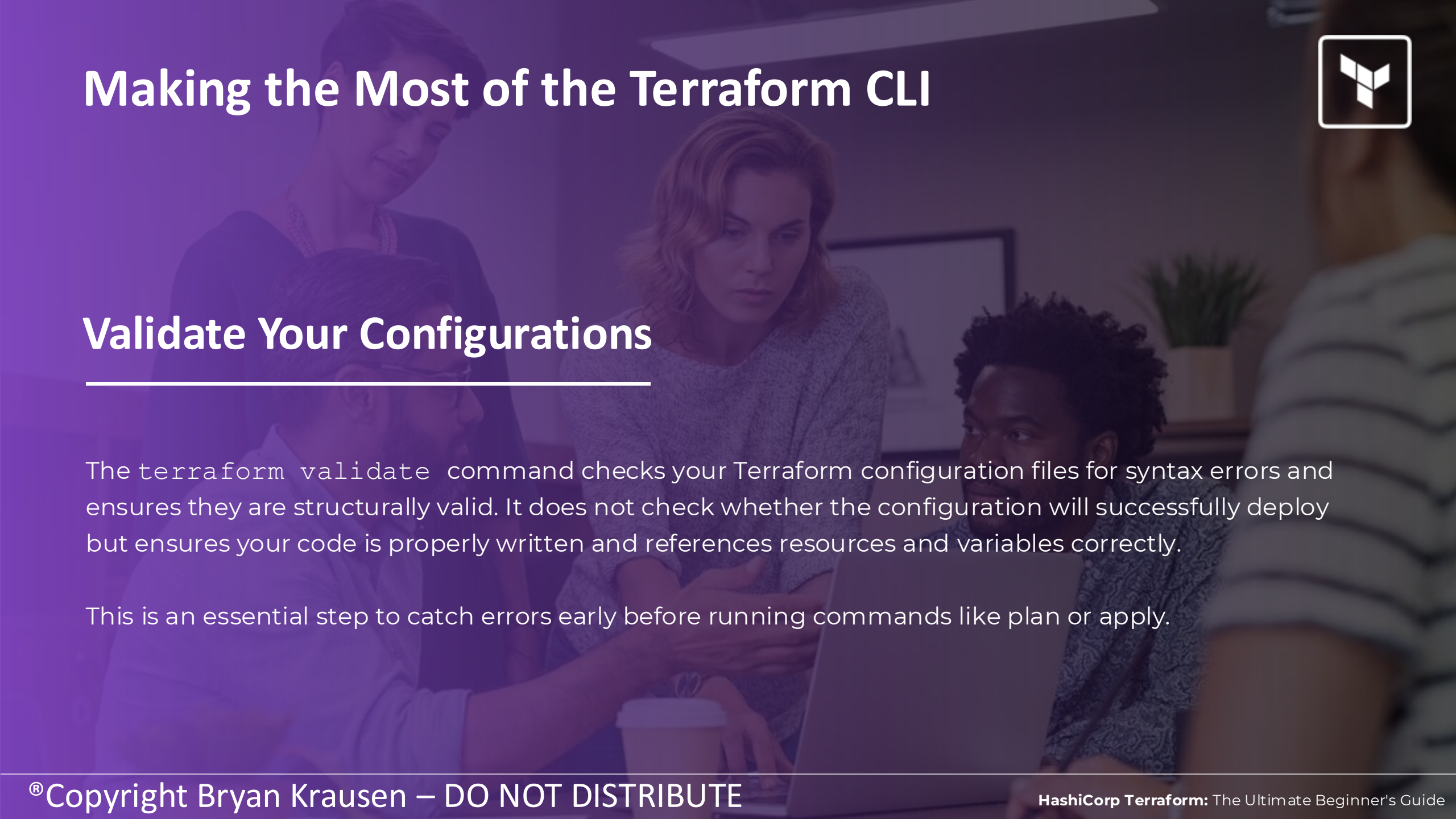


`terraform fmt`

Format all Terraform files in the current directory

Use `-recursive` to format files in subdirectories as well

Making the Most of the Terraform CLI



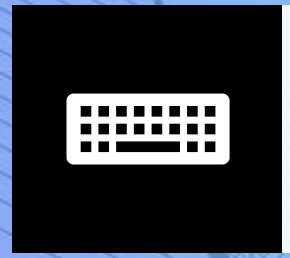
Validate Your Configurations

The `terraform validate` command checks your Terraform configuration files for syntax errors and ensures they are structurally valid. It does not check whether the configuration will successfully deploy but ensures your code is properly written and references resources and variables correctly.

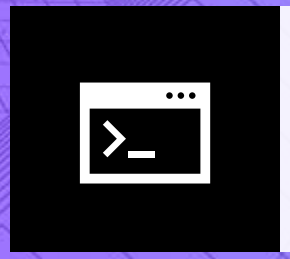
This is an essential step to catch errors early before running commands like plan or apply.

SUMMARY

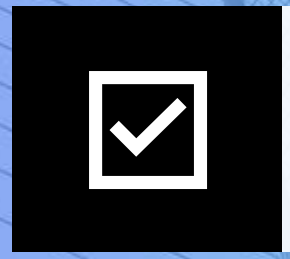
Use `terraform <subcommand>` to manage each step of the workflow



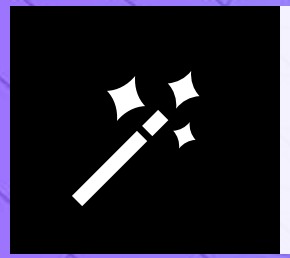
Enable `terraform -install-autocomplete` to speed up typing and reduce errors



Customize commands with options and flags depending on the subcommand you're using.



Run `terraform validate` to catch syntax errors early



Use `terraform fmt` to ensure clean and consistent formatting

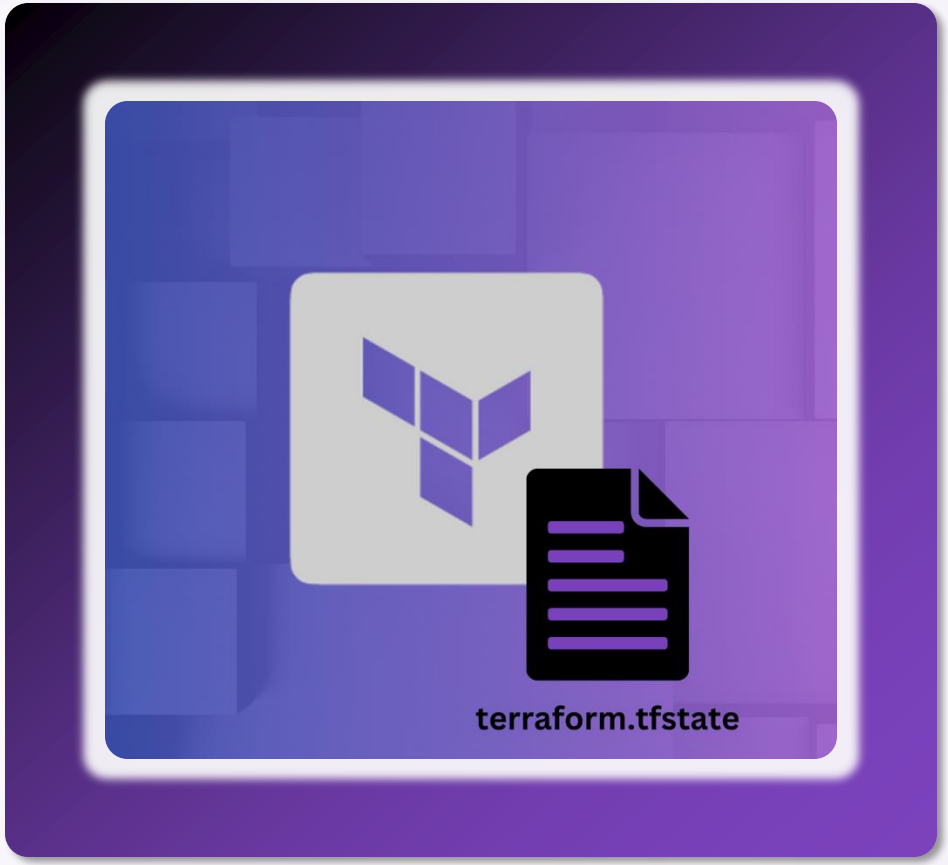
Pass sensitive data and reusable defaults securely via `TF_VAR_*` and other variables.

LECTURE

Understanding Terraform State

Introduction to Terraform State

Terraform state is a file that tracks the resources managed by Terraform, including their IDs and relationships.

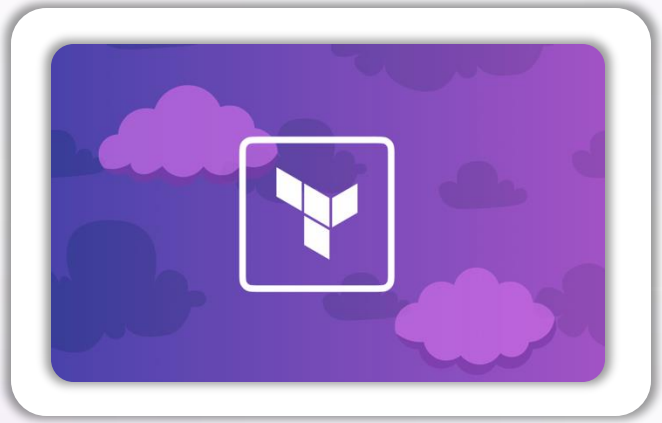


- It acts as a single source of truth for your infrastructure. Without the state file, Terraform would not know its current state.
- Terraform cannot operate without state
- Before any operation, Terraform does a refresh to update the state with the real-world infrastructure

Why Terraform State is Important

Resource Management

Enables Terraform to determine the difference between the desired and current state.



Dependency Management

Tracks resource relationships to apply updates in the correct order.

Terraform does a refresh to update the state before any operations

Code Collaboration

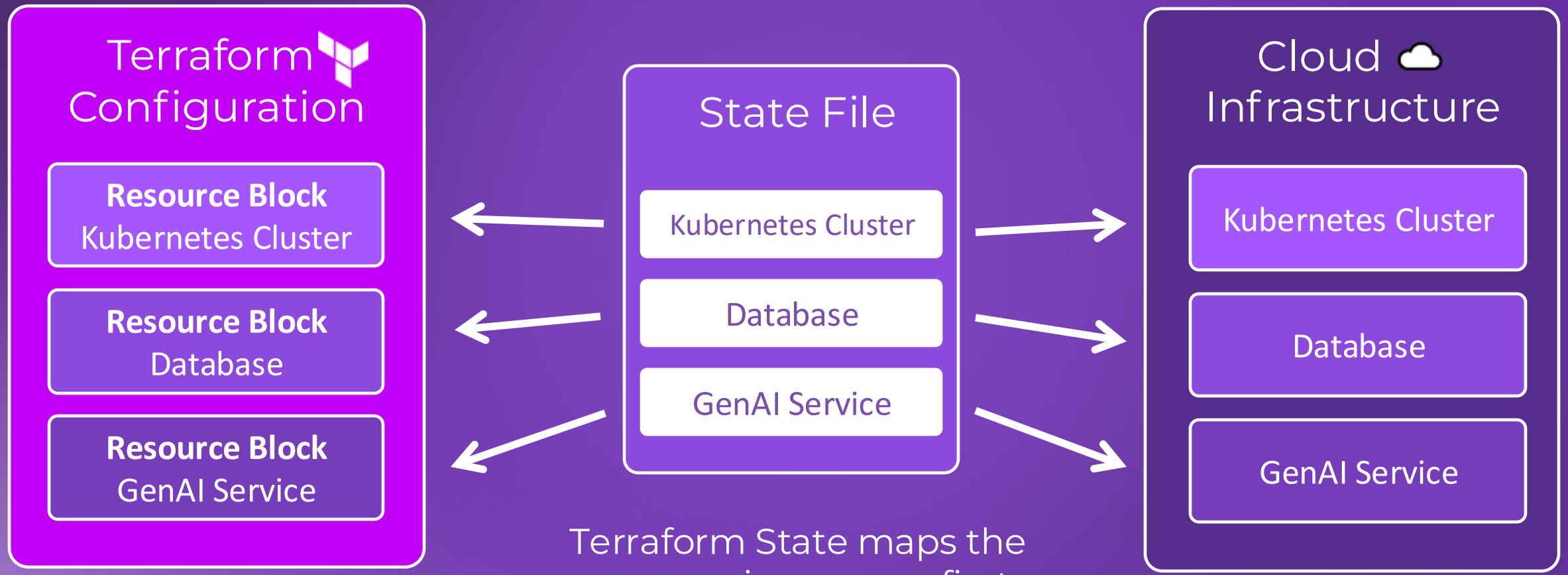
Remote state enables secure, shared workflows for teams.

State File

- Terraform stores state in file, which acts as a sort of database to map Terraform configurations to real-world resources
- Alongside the mapping between resources and remote objects, Terraform also tracks metadata for internal purposes.
- This metadata allows Terraform to track order of operations, the provider used, etc.

```
{
  "module": "module.nomad-client[\"nomad-client-3\"]",
  "mode": "managed",
  "type": "aws_instance",
  "name": "this",
  "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
  "instances": [
    {
      "index_key": 0,
      "schema_version": 1,
      "attributes": {
        "ami": "ami-03a6eaae9938c858c",
        "arn": "arn:aws:ec2:us-east-1:634211456147:instance/i-0eb4591447d4fdf7c",
        "associate_public_ip_address": true,
        "availability_zone": "us-east-1c",
        "capacity_reservation_specification": [
          {
            "capacity_reservation_preference": "open",
            "capacity_reservation_target": []
          }
        ],
        "cpu_core_count": 1,
        "cpu_options": [
```

Terraform State



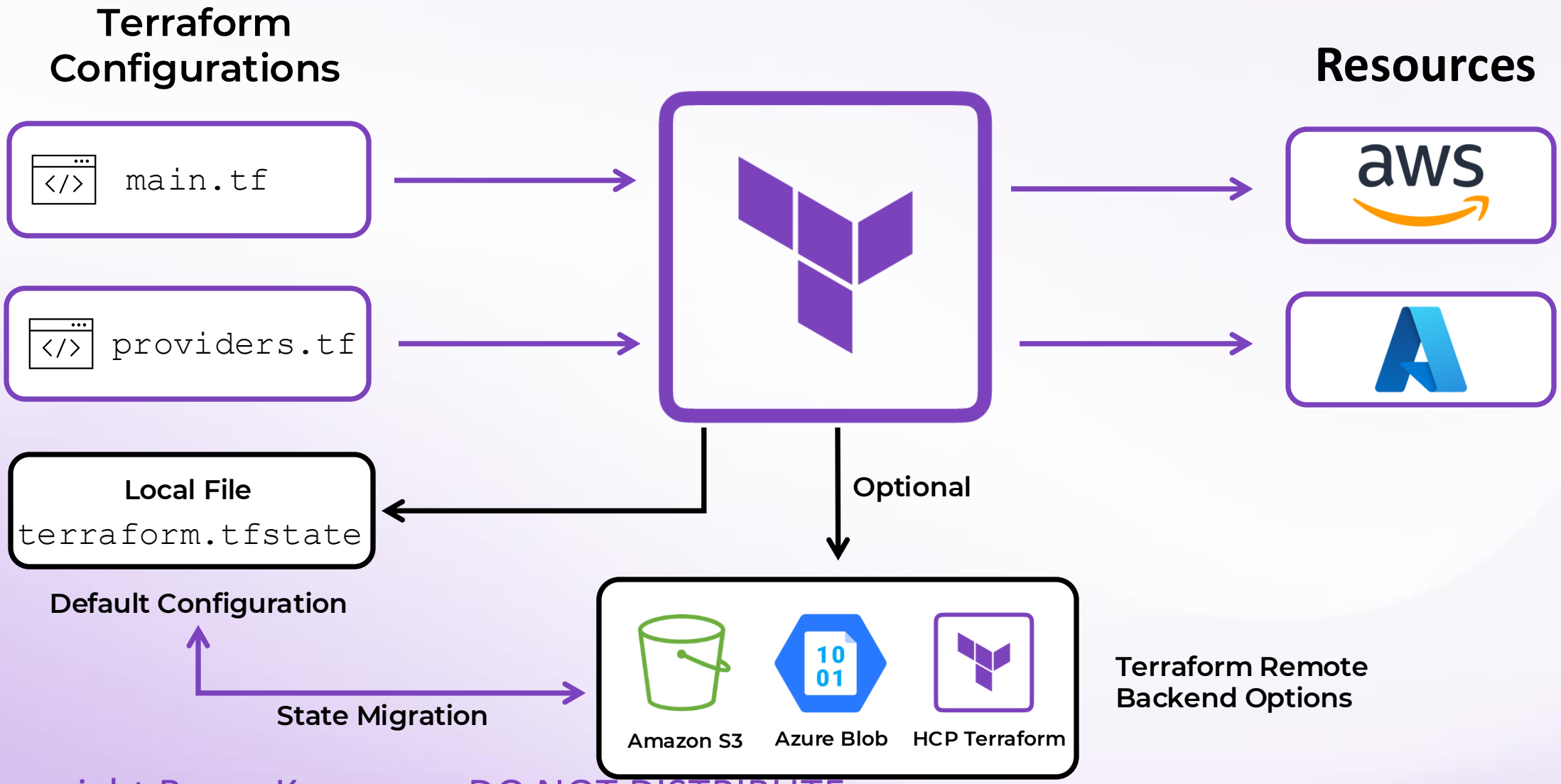
Terraform State maps the resources in your config to the real-world infrastructure

State Storage

- By default, Terraform stores state in the current working directory in a JSON-formatted file named `terraform.tfstate`
- Terraform needs to "lock" the state file when performing any operations that might write to the file (to avoid conflicts)
- Terraform can be configured to store state in a shared location. A backend configuration will determine where state is stored

```
{
  "module": "module.nomad-client[\"nomad-client-3\"]",
  "mode": "managed",
  "type": "aws_instance",
  "name": "this",
  "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
  "instances": [
    {
      "index_key": 0,
      "schema_version": 1,
      "attributes": {
        "ami": "ami-03a6eaae9938c858c",
        "arn": "arn:aws:ec2:us-east-1:634211456147:instance/i-0eb4591447d4fdf7c",
        "associate_public_ip_address": true,
        "availability_zone": "us-east-1c",
        "capacity_reservation_specification": [
          {
            "capacity_reservation_preference": "open",
            "capacity_reservation_target": []
          }
        ],
        "cpu_core_count": 1,
        "cpu_options": [
```

Terraform – Remote State



LECTURE

Making Code Reusable

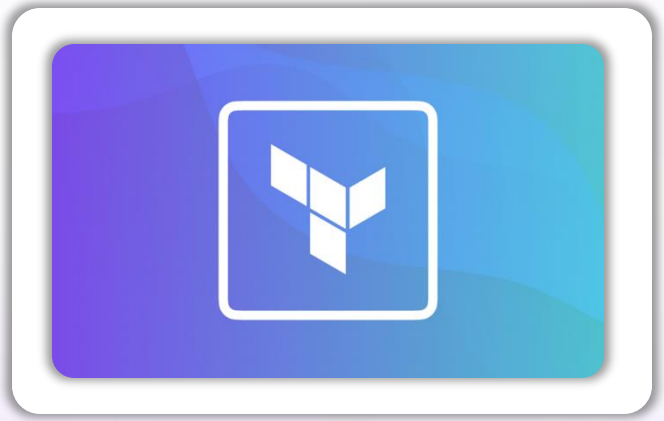
The "Why" Behind Reusability

Building scalable, maintainable infrastructure starts with reusing and standardizing your code. This ensures you can adapt quickly and minimize errors as your projects grow.

- Eliminating repetition (the DRY principle) keeps your code lean and updates simple.
- Centralizing common values reduces confusion and ensures consistency.
- Shared patterns help teams collaborate more effectively and onboard new members quickly.

LET'S MAKE YOUR CODE BETTER

The DRY Principle (Don't Repeat Yourself)



Minimize Repeated Code

Copying similar blocks of code can create confusion and tech debt.

DRY keeps your Terraform files clean and easy to maintain

Centralize Common Logic

Using variables, locals, or modules allow you to define values or resource patterns once and reuse them over and over again

Real-World Benefits

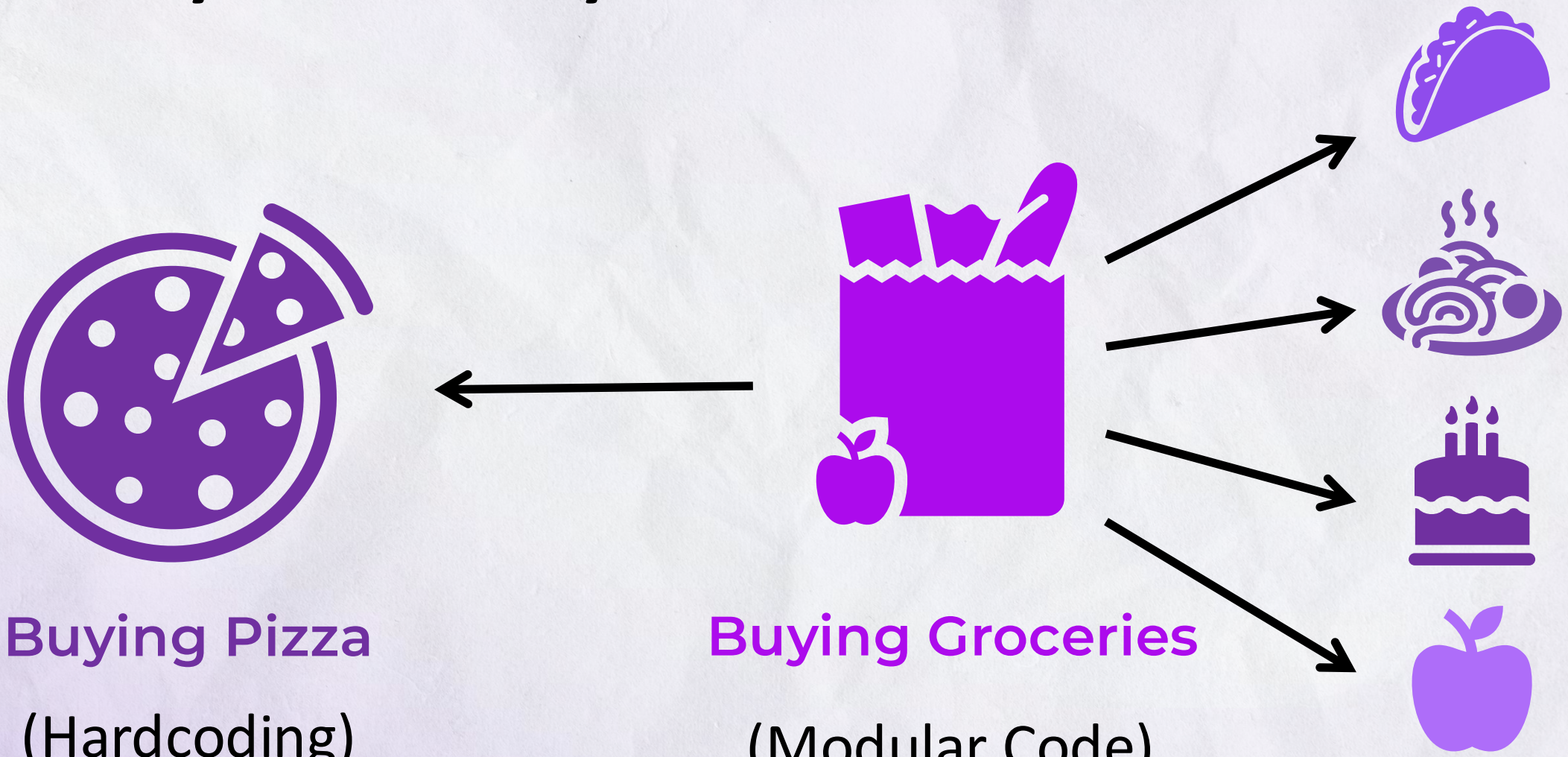
Helps standardize names, tags, or resources across the board.

A single source of truth reduces errors and speeds up changes.

The Benefits of Reusable Code

- **Maintainability** – Easier to update and debug
- **Scalability** – Quickly extend or modify infrastructure
- **Consistency** – Uniformity across resources minimizes errors
- **Collaboration** – A clear structure makes teamwork simpler

Reusability = Flexibility



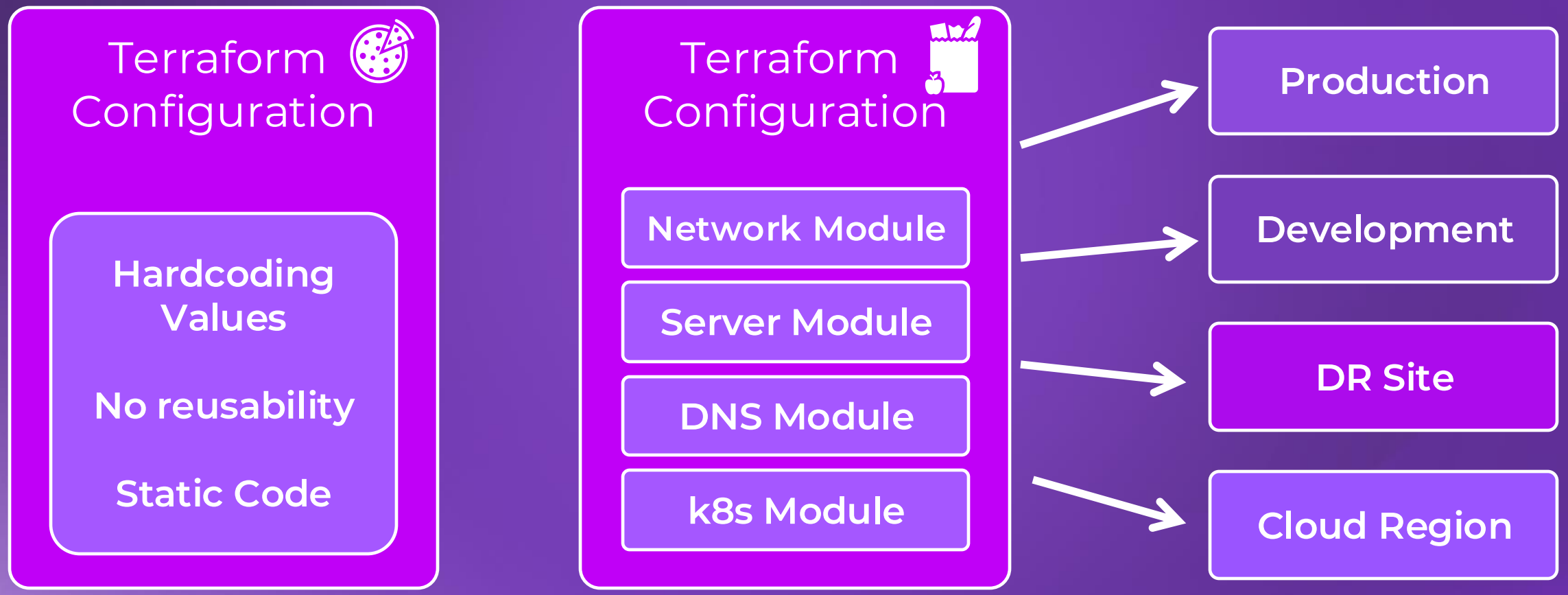
Buying Pizza

(Hardcoding)

Buying Groceries

(Modular Code)

Repeatability = Flexibility



LECTURE

Enhancing Code with Dynamic Values

Variables Recap

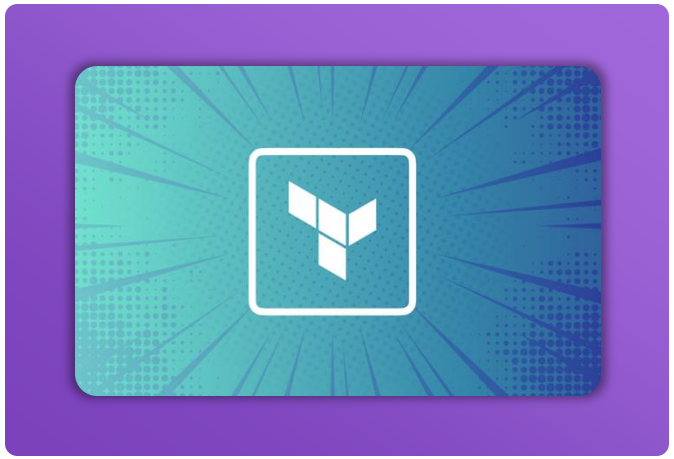
Dynamic Inputs for Terraform Code

Input variables let you customize Terraform configurations without changing the source code or hardcoding values

Variable Types

Variable types include string, number, bool, map, set, list.

Set values via defaults, ENV, .tfvars, and command line



Variables Keep Your Code DRY

Pass in different values to create unique resources

Source: HashiCorp State of Cloud Strategy Survey

Data Sources - Reap

Pull External Information into Terraform

Provider-Specific Info

Data sources (blocks) query provider-specific information about existing resources so we can use it in our code

Common Use Cases

Dynamic lookups, reference existing infrastructure, and avoid hardcoding or repeating information

No Changes

Data sources don't create or make any changes to infrastructure. They only read information and pull it into Terraform

Combining Variables and Data Sources Can Create More Flexible Code

Dynamic code doesn't use static values – it uses interpolation, meta-arguments, functions, local values, and other methods to adapt to changing needs.

Static Inputs & Variables

main.tf

```
resource "aws_instance" "example" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"
}

resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
  tags = {
    Name = "hardcoded-vpc"
  }
}
```

main.tf

```
resource "github_repository" "example" {
  name        = "my-static-repo"
  description = "A static name"
  visibility  = "public"
}

resource "github_team" "example" {
  name        = "hardcoded-team"
  description = "A team with a name"
}
```

Dynamic Code

```
resource "aws_instance" "example" {
  ami = data.aws_ami.ubuntu.id
  instance_type = local.instance_type

  tags = {
    Name = "${var.app}-${local.env}-server"
  }
}

resource "aws_vpc" "example" {
  cidr_block = var.vpc_cidr

  tags = {
    Name = "vpc-${local.env}-${data.aws_region}-${var.network}"
  }
}
```

Interpolation

This ${ . . . } sequence is called an interpolation. It evaluates the expression between the markers, converts the result, and inserts it into the string

main.tf

```
variable "name" {
  type    = string
  default = "Bryan"
}


"Hello, ${var.name}!"
Hello, Bryan!


"My name is ${var.name}, I'm your instructor!"
My name is Bryan, I'm your instructor!
```

Interpolation

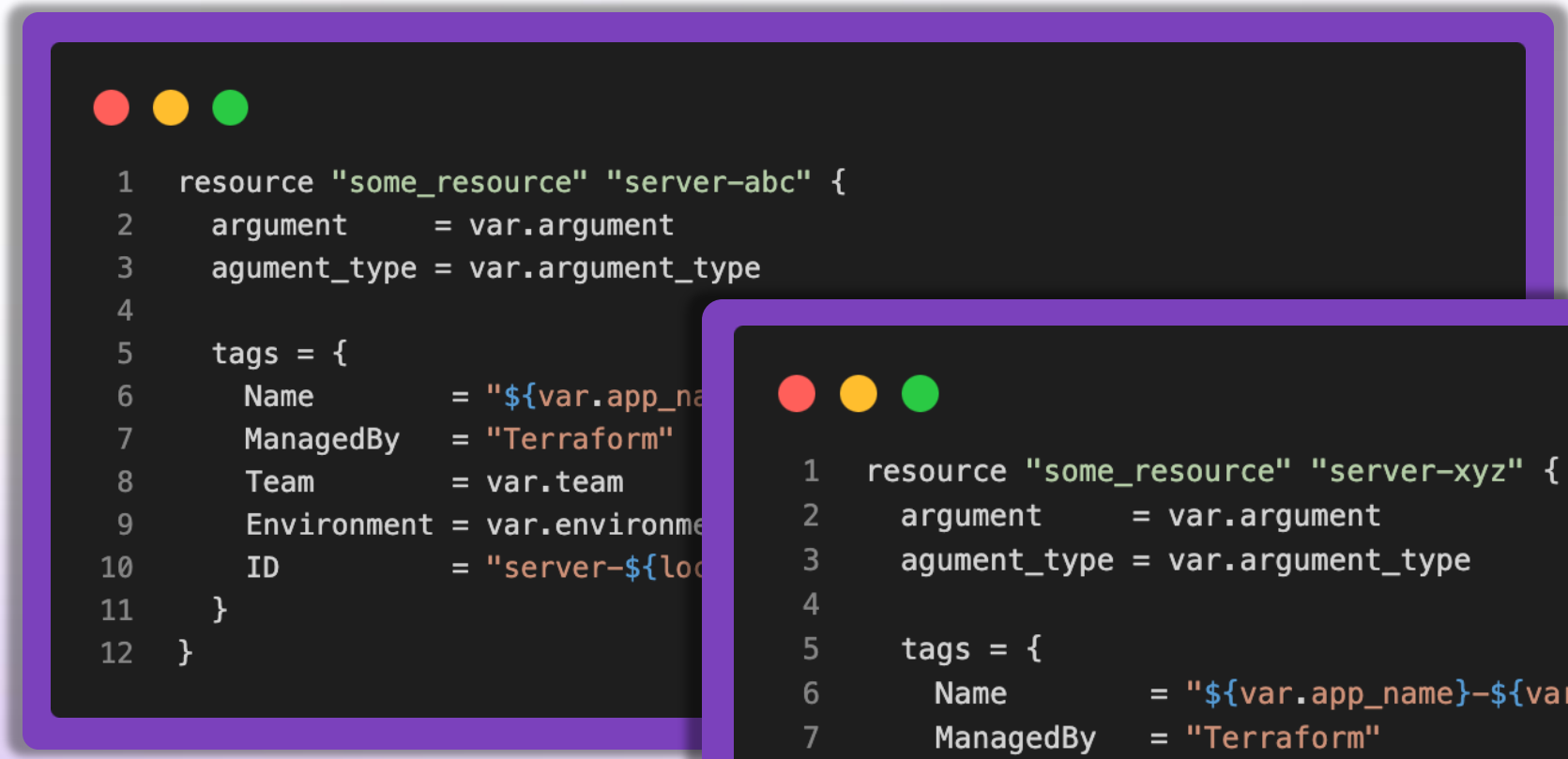
main.tf

```
variable "environment" {
  type    = string
  default = "dev"
}

data "aws_region" "current"{}
data "aws_caller_identity" "current" {}


name = "server-${data.aws_region.current.name}-${var.environment}"
server-us-east-1-dev

bucket_name = "s3-${data.aws_caller_identity.current.account_id}-backups"
s3-9876543210-backups
```

LECTURE

Using Locals to Avoid Code Duplication

When Code Duplication Strikes



```
resource "some_resource" "server-abc" {
  argument     = var.argument
  agument_type = var.argument_type

  tags = {
    Name        = "${var.app_na
    ManagedBy   = "Terraform"
    Team        = var.team
    Environment = var.environme
    ID          = "server-${loc
  }
}
```

```
resource "some_resource" "server-xyz" {
  argument     = var.argument
  agument_type = var.argument_type

  tags = {
    Name        = "${var.app_name}-${var.server}-xyz"
    ManagedBy   = "Terraform"
    Team        = var.team
    Environment = var.environment
    ID          = "server-${local.env}-${data.aws_region}-${var.server[1]}"
  }
}
```

What are Local Values?

Named values that you can reference in other parts of your Terraform code

- Most often referred to as "locals," think of locals like short-term memory for your Terraform code
- They centralize and store values you use often, like names, environments, or shared tags
- Local values can be helpful to avoid repeating the same values or expressions multiple times in a configuration

```
1  locals {
2    app_name    = "my-app"
3    environment = "dev"
4  }
5
```

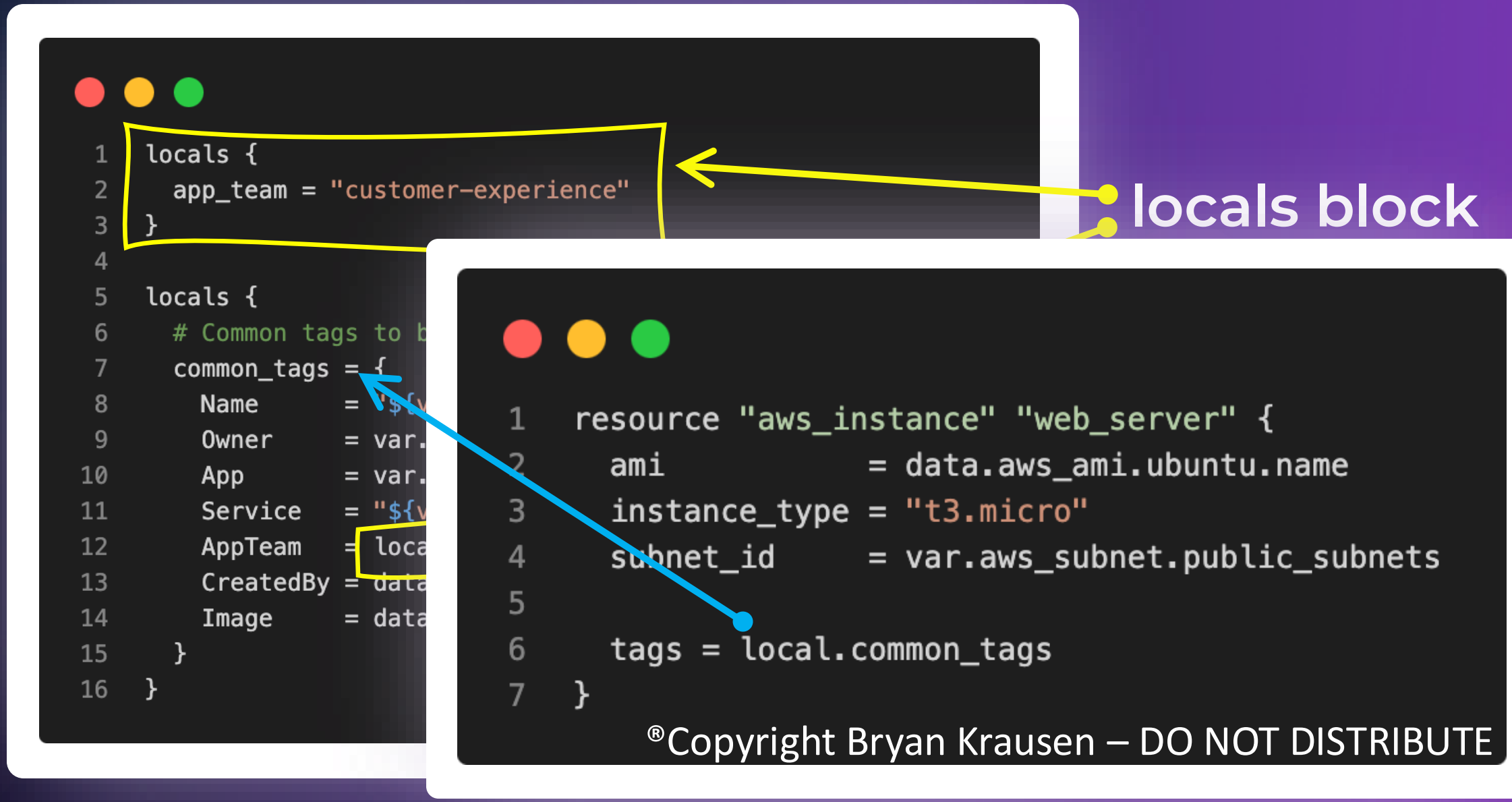
Basic Syntax - Local Values

```hcl
# Define shared values in locals
locals {
  environment = "dev"
  prefix      = "myorg"
}

resource "github_repository" "dev-repo" {
  name        = "${local.prefix}-${local.environment}-repo"
  visibility  = "private"
  description = "Terraform-managed repo for dev environment"
}

resource "github_team" "awesome-people" {
  name        = "${local.prefix}-${local.environment}-team"
  description = "My Awesome Team"
  privacy     = "closed"
}
```

locals block

Using a locals block

Basic Syntax - Local Values



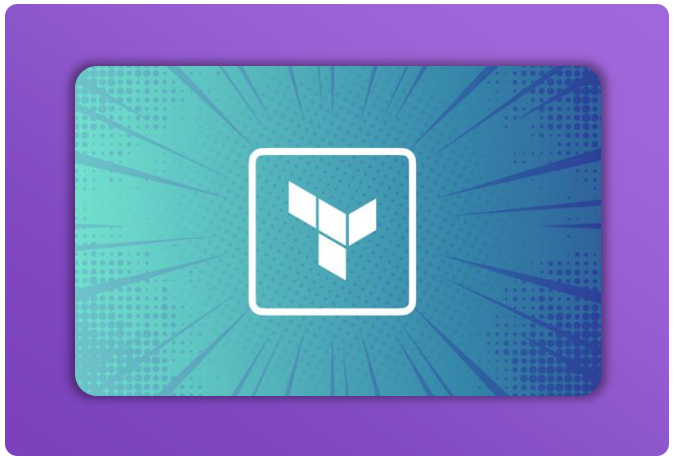
Benefits of Locals

Centralized Logic

The ability to easily change the value in a central place is the key advantage of local values

Clearer Code

Reduces clustter and makes it obvious where a value comes from



Less Risk

Changing a local is less error-prone than updating multiple hardcoded values

When Should I Use Local Values?

- When you need to combine variables or data sources to create dynamic values, locals help keep your code readable and consistent.

- If the same value appears multiple times (like a prefix, environment name, or common tag), define it once in locals instead of repeating it across multiple resources.

- Instead of embedding lengthy calculations or conditions directly in resource definitions, you can compute them in locals first and reference them where needed.

LECTURE

Leveraging Meta-Arguments to Reduce Repetition

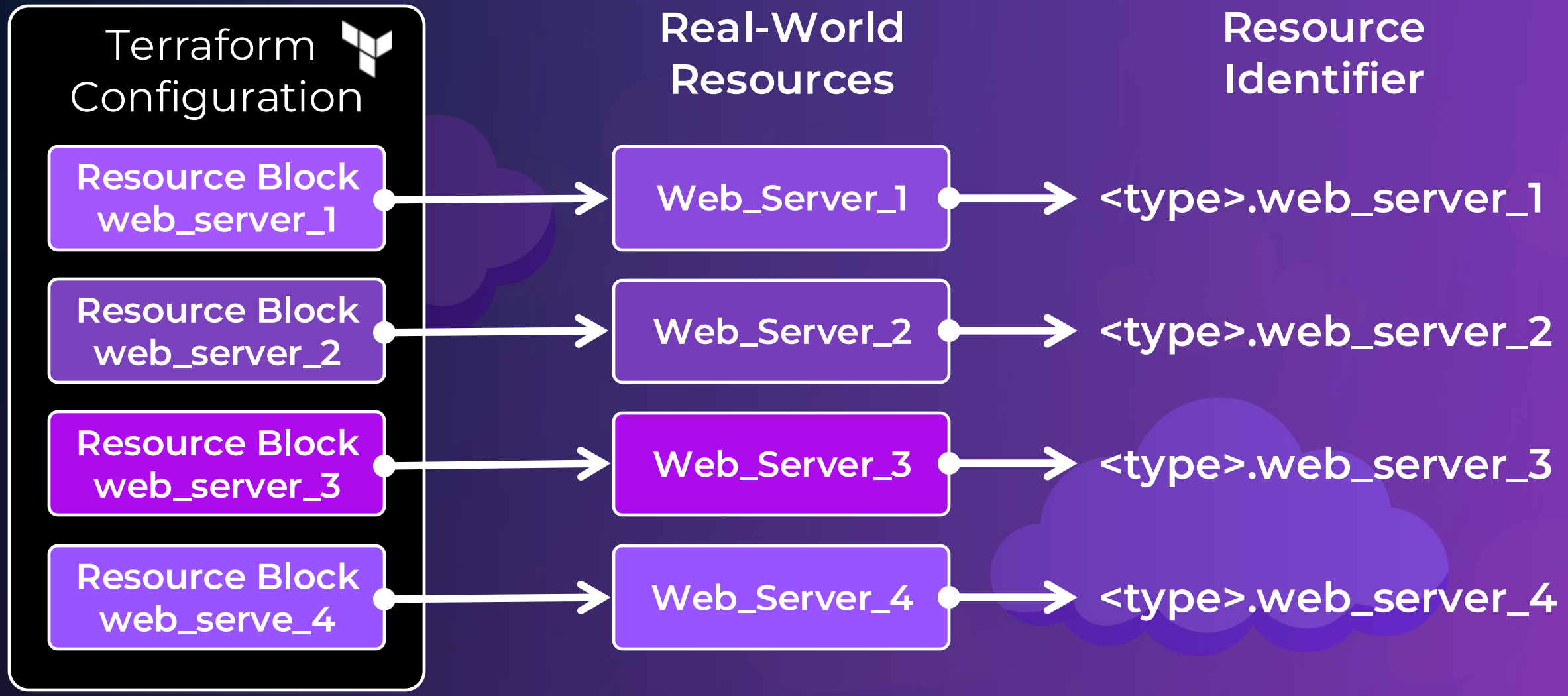
What are Meta-Arguments?

Meta-arguments are special arguments that modify how Terraform creates, manages, and structures multiple resources.

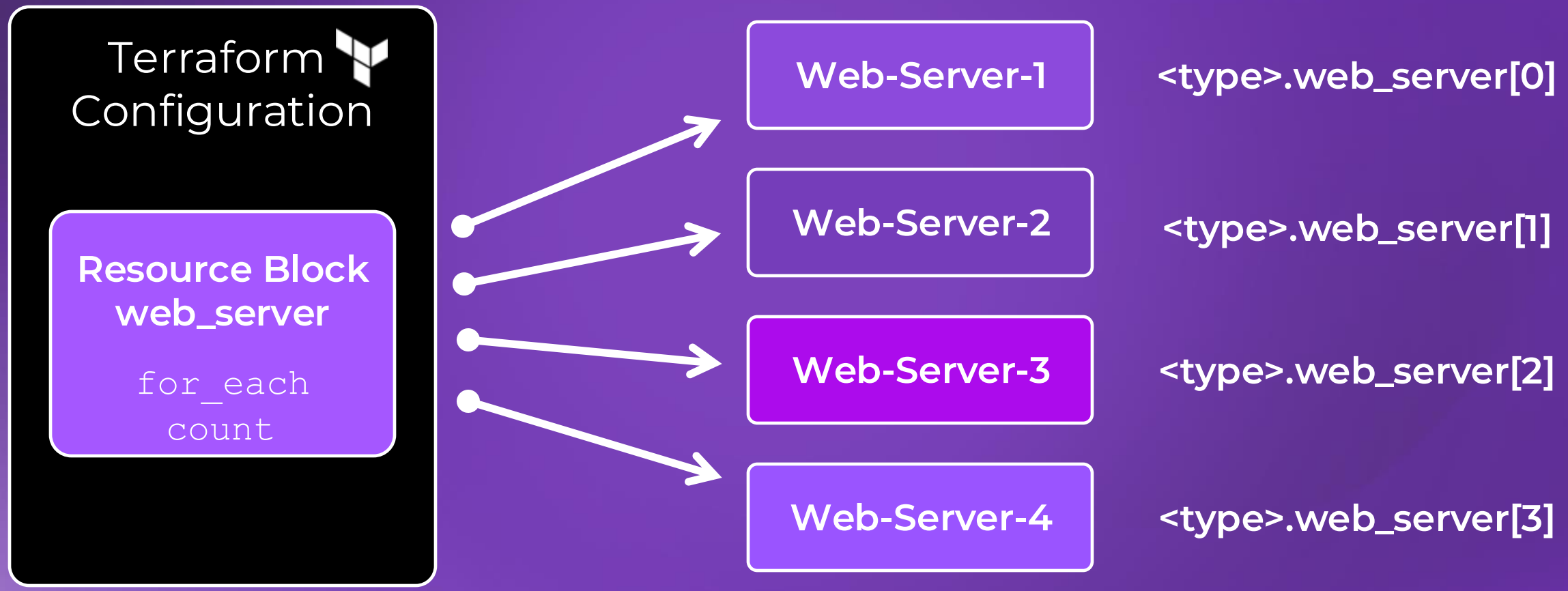
- Reduce redundancy
- Make configurations more scalable
- Simplify resource dependencies

Think of meta-arguments as Terraform's built-in automation tools. Instead of copy-pasting similar resources, you can use meta-arguments to create them dynamically based on input data.

Don't Repeat Yourself



Use Meta-Arguments Instead



Meta-Arguments

Write Less Code, Deploy More Efficiently

`count`

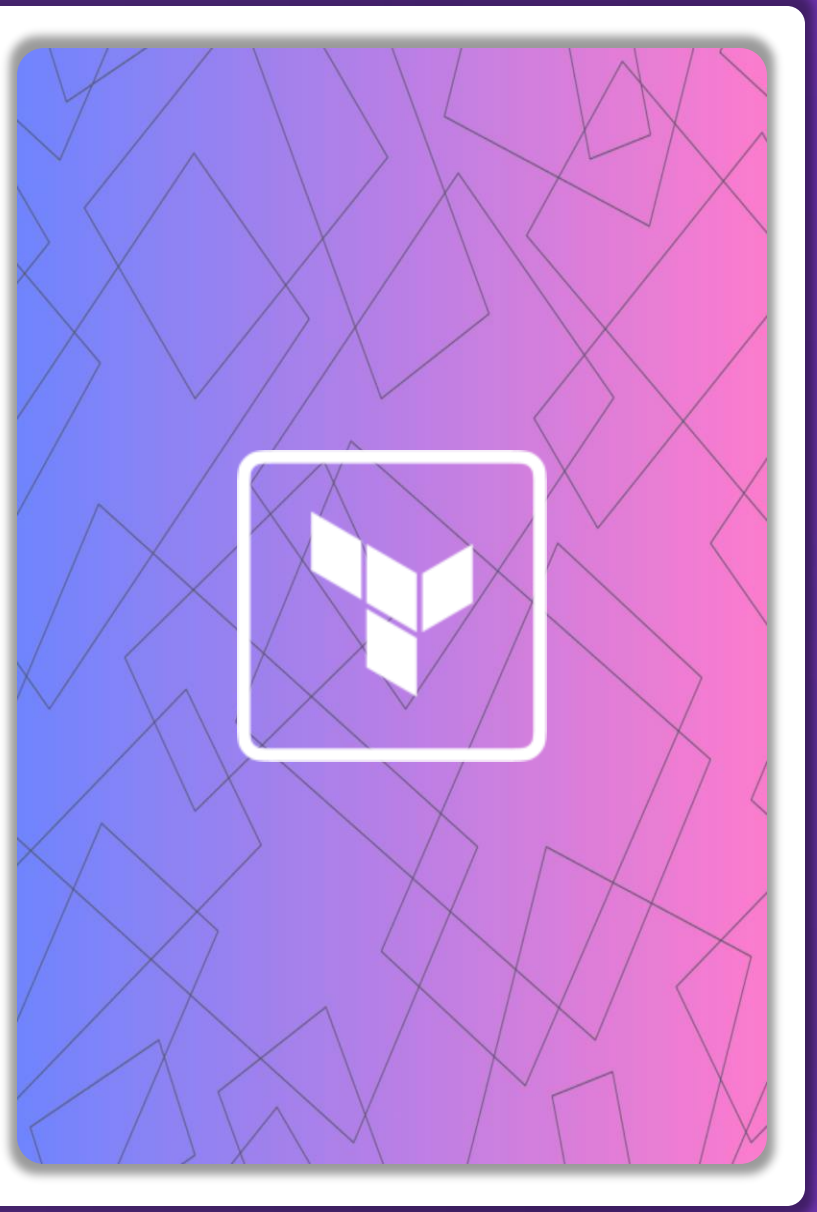
Allows you to specify a number of instances for a resource.
Example: Deploying five virtual machines with the same configurations

`for_each`

Lets you create multiple resources dynamically from a map or set.
Example: Deploying three VMs with different names instead of the same

`depends_on`

Explicitly define that a resource must be created before another.
Example: Ensure a database is provisioned before an app server



Meta-Arguments

Write Less Code, Deploy More Efficiently

`provider`

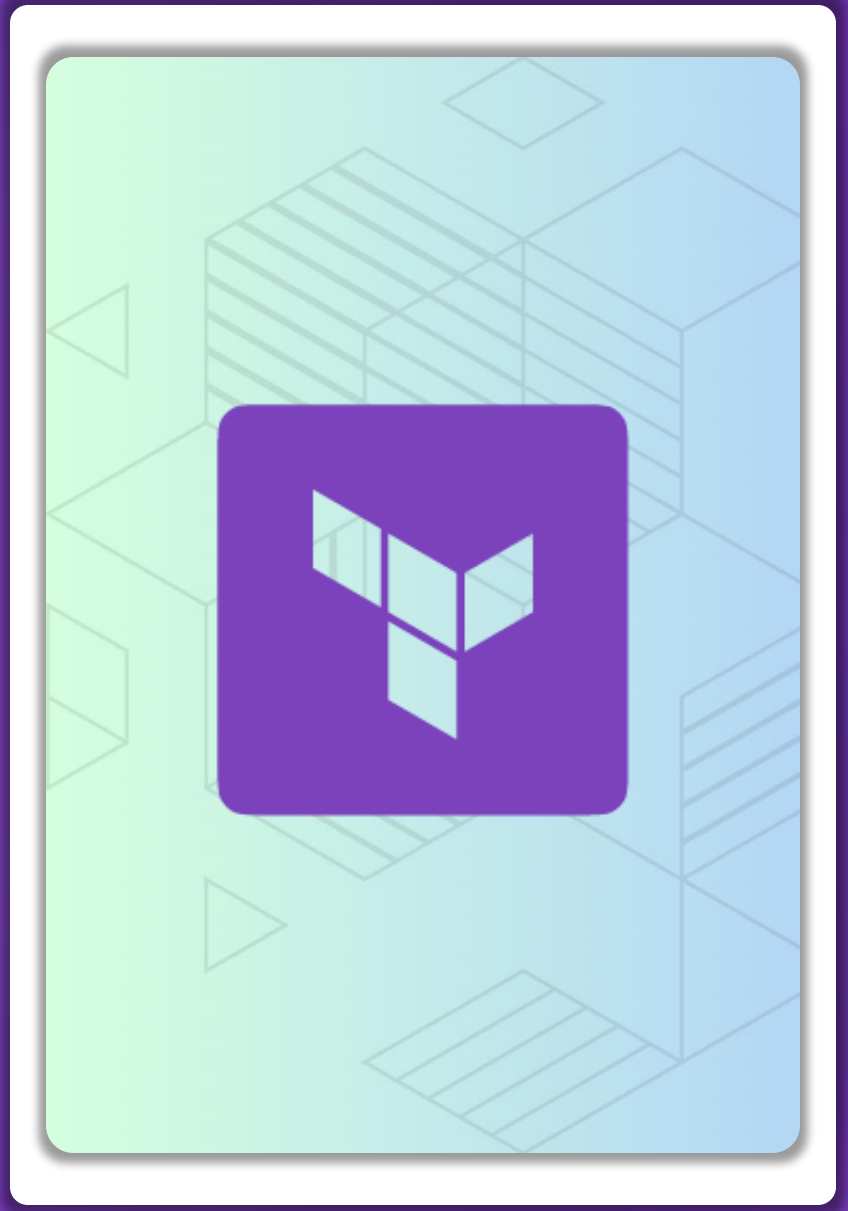
Specify what provider configuration to use

Example: Assign resources to use a different provider block mapped to a second region

`lifecycle`

Control how Terraform manages the lifecycle of the resource

Example: Protect critical infrastructure or ensure that Terraform doesn't replace resources unnecessarily



`count` Meta-Argument

Specify a number of instances for a resource

- Allows Terraform to create multiple identical resources using a single block
- Assigns an index (starts at 0) to reference each instance
- Use `count` to create resources that are identical but need to be repeated or when a variable controls the number of resources
- Reference the resource using `<type>.<name>[index]`

```hcl
variable "vm_count" {
  type    = number
  default = 3
}

resource "azurerm_virtual_machine" "example" {
  count                 = var.vm_count
  name                  = "vm-${count.index}"
  location              = "East US"
  resource_group_name = "my-resource-group"
  vm_size               = "Standard_D2s_v3"

  os_profile {
    computer_name  = "vm-${count.index}"
    admin_username = "adminuser"
  }
}
```

Variable controlling the number of resources

`azurerm_virtual_machine.example[0]`

`for_each` Meta-Argument

A More Flexible Way to Manage Multiple Resources

Creates Multiple Resources

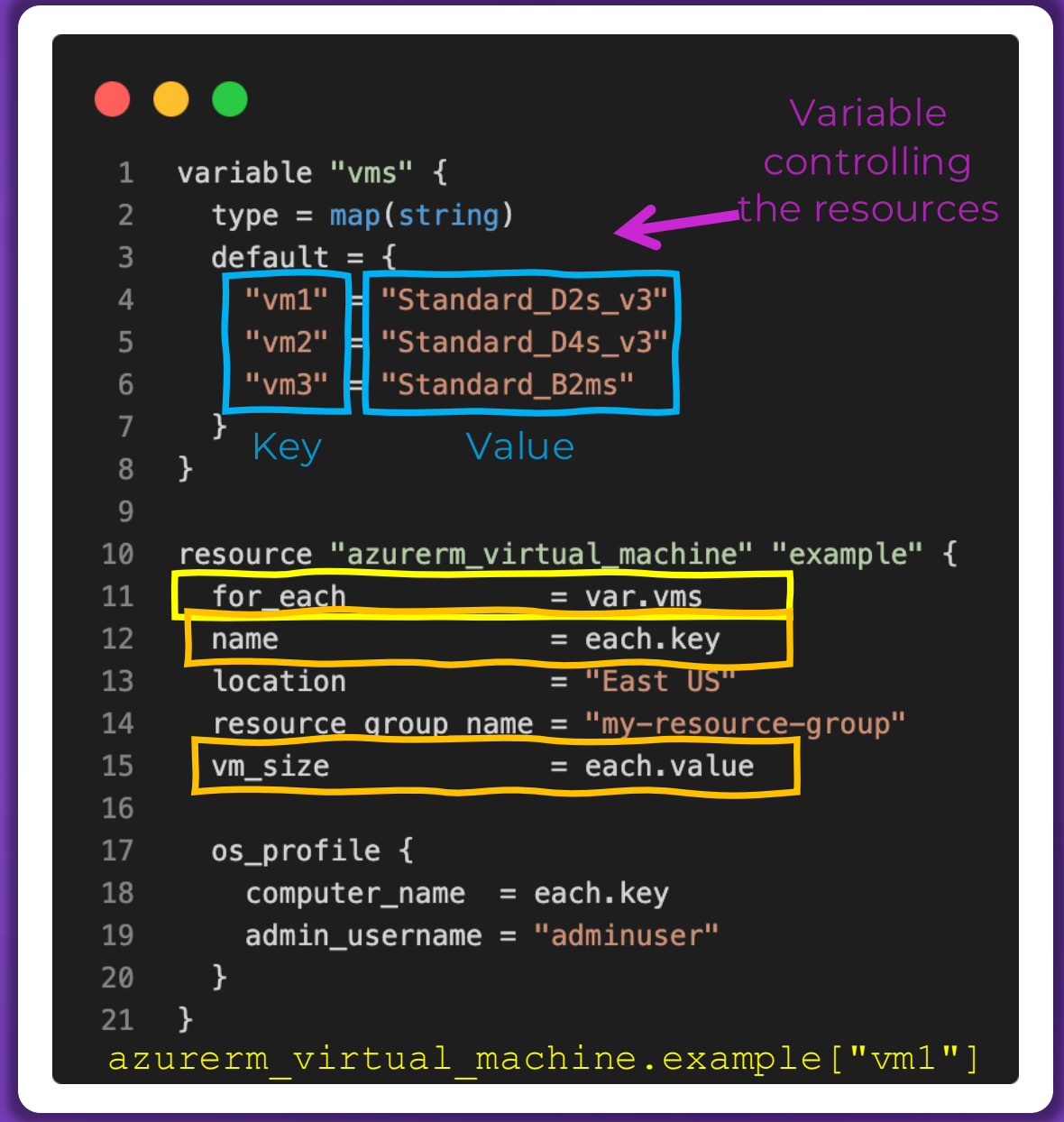
Pulls in values from a map or set rather than a fixed number

Resources Get a Unique Key

Rather than an index, each resource gets a unique key as a reference, making it more stable than count

Ideal for Managing Unique Configurations

Use it when resources need different names, tags, or settings




```
variable "vms" {
  type = map(string)
  default = {
    "vm1" = "Standard_D2s_v3"
    "vm2" = "Standard_D4s_v3"
    "vm3" = "Standard_B2ms"
  }
}

resource "azurerm_virtual_machine" "example" {
  for_each            = var.vms
  name                = each.key
  location            = "East US"
  resource_group_name = "my-resource-group"
  vm_size             = each.value

  os_profile {
    computer_name  = each.key
    admin_username = "adminuser"
  }
}
azurerm_virtual_machine.example["vm1"]
```

depends_on Meta-Argument

Ensuring Resources Are Created in the Right Order

Explicitly Define Dependencies

Terraform may not automatically detect or know about dependencies needed for your deployment

Ensure Correct Provisioning Order

Useful for resources that rely on others to exist first

Prevents Race Conditions

Complex deployments may require one resource to be created before another

```
resource "aws_db_instance" "db" {
  identifier          = "mydb"
  engine              = "mysql"
  instance_class      = "db.t3.micro"
  allocated_storage   = 20
  username            = "admin"
  password            = "SuperSecurePassword123!"
  skip_final_snapshot = true
}

resource "aws_instance" "app_server" {
  ami           = "ami-0abcdef1234567890"
  instance_type = "t2.micro"

  depends_on = [aws_db_instance.db]
}
```

`provider` Meta-Argument

Explicitly Assign Resources to a Specific Provider

- Explicitly assigns a resource to a specific provider configuration.
- Most often used for multi-cloud or multi-account deployments.
- Use it when deploying resources across multiple cloud accounts or regions or managing various instances of the same provider (*e.g., separate AWS accounts for dev and prod*).
- Uses the `provider = <provider_alias> syntax`

```
1  provider "aws" {
2    region = "us-east-1"
3  }
4
5  provider "aws" {
6    alias  = "prod"
7    region = "us-west-1"
8  }
9
10 resource "aws_s3_bucket" "dev_bucket" {
11   bucket   = "my-dev-bucket"
12 }
13
14 resource "aws_s3_bucket" "prod_bucket" {
15   provider = aws.prod
16   bucket   = "my-prod-bucket"
17 }
```

`lifecycle` Meta-Argument

Controlling How Terraform Manages Resources

- **Customizes how Terraform Manages Resources**

  Change the default behavior during your Terraform workflow.

- **You can protect resources or avoid changes**

  Customizes how Terraform creates, updates, and destroys specific resources.

- **Uses the `lifecycle` block inside of your resource**

  Options include `prevent_destroy`, `create_before_destroy`, and `ignore_changes`

```hcl
resource "azurerm_resource_group" "example" {
  name     = "my-resource-group"
  location = "East US"

  lifecycle {
    prevent_destroy = true
  }
}

resource "azurerm_virtual_machine" "example" {
  name                = "my-vm"
  location            = "East US"
  resource_group_name = "my-resource-group"
  vm_size             = "Standard_D2s_v3"

  lifecycle {
    ignore_changes = [vm_size]
  }
}
```

LECTURE

Using Built-In Functions to Standardize Code

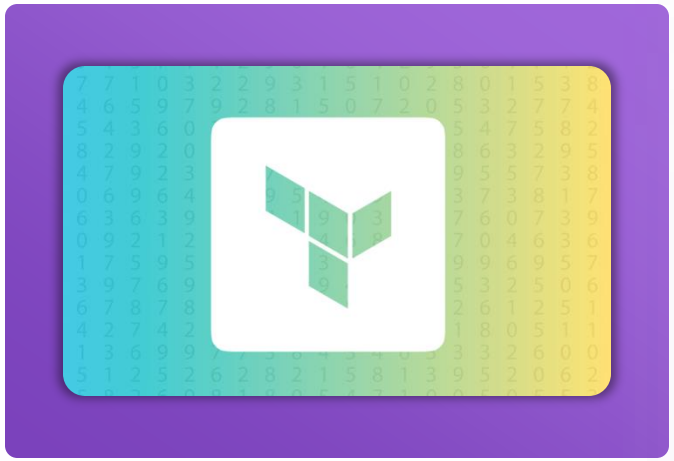
Built-In Functions

Simplify infrastructure code by providing built-in helpers to manipulate strings, numbers, lists, and more.

- Reduces repetitive tasks
- Simplifies logic
- Ensures consistent patterns in deployments

Mastering core functions creates more maintainable, reliable, and readable Terraform configurations.

Core Terraform Functions



Numeric Functions

Quickly evaluate numeric values (e.g., finding the smallest subnet size).

Examples:

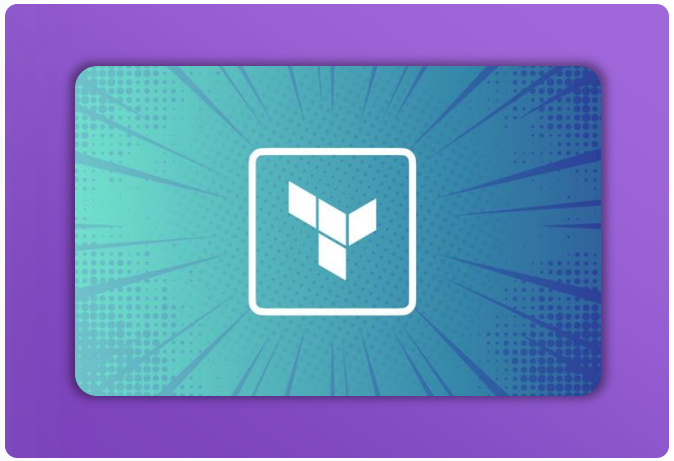
`mix` **&** `max`

String Functions

Concatenate or encode data for resource names, user data, etc.

Examples:

`join` **&** `split`



Type Conversions

Ensure consistent data structures across your configuration and avoid type mismatch issues.

Examples:

`toset` **&** `tolist`

General Syntax for Terraform Built-In Functions

Syntax: `function_name(argument1, argument2, ...)`

- **Example 1:** `upper("hello")` → **returns** `"HELLO"`
- **Example 2:** `min(4,7,2,9,5)` → **returns** `2`
- **Example 3:** `join("-", "hello", "terraform")` → `"hello-terraform"`

Numeric Functions

Determining the largest or smallest value among different variables/outputs

- **Dynamic Resource Allocation:** Adjust CPU, memory, or storage sizes without hardcoding.
- **Automated Calculations:** Remove guesswork by letting Terraform handle numeric logic.

```
# Max: Returns the highest value in a list of numbers
max(10, 4, 7) → 10

# Min: Returns the lowest value in a list of numbers
min(10, 4, 7) → 4

# ####################

variable "number" {
  type    = number
  default = 15
}

max(10, 4, var.number) → 15
```

String Functions

Manipulate and transform textual data — enabling consistent naming, simplified automation, and more maintainable configurations.

```
# Join: Concatenates a list of strings into a single string.
join("-", ["prod", "web", "us-west-1"]) → prod-web-us-west-1

# upper / lower: Purpose: Converts a string to uppercase/lowercase
upper("example") → EXAMPLE
lower("HELLO") → hello

# replace: Substitutes part of a string with another value.
replace("123-abc", "abc", "xyz") → 123-xyz

# base64encode: Encodes string to Base64, often required for user data
base64encode("my-secret-data") → "bXktc2VjcmV0LWRhdGE="
```

Network Functions

Specialized functions to work with IP addresses and subnets

- **Automate CIDR Calculations:** Build subnet blocks and host addresses seamlessly.
- **Reduce Manual Errors:** Eliminate hand-calculating IP ranges.
- **Scalable Network Design:** Easily create multiple subnets in large or complex deployments.

```hcl
# Create a VPC with a /16 CIDR
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}

# Create a public subnet in the first available range
resource "aws_subnet" "public" {
  vpc_id            = aws_vpc.main.id
  availability_zone = "us-east-1a"
  cidr_block        = cidrsubnet(aws_vpc.main.cidr_block, 3, 0)
}

# Create a private subnet in the second available range
resource "aws_subnet" "private" {
  vpc_id            = aws_vpc.main.id
  availability_zone = "us-east-1b"
  cidr_block        = cidrsubnet(aws_vpc.main.cidr_block, 3, 1)
}
```

Type Conversion Functions

Convert collections of data into different types for inputs or outputs

- `toset` / `tolist`
  Convert collections into lists or sets
- `tomap`
  Convert a list or object to a map
- `tostring`
  Convert the argument to a string

```hcl
variable "availability_zones" {
  type    = list(string)
  default = ["us-east-1a", "us-east-1a", "us-east-1b"]
}

locals {
  unique_zones = toset(var.availability_zones)
}

resource "aws_subnet" "example" {
  for_each = local.unique_zones

  vpc_id            = aws_vpc.main.id
  availability_zone = each.value
  cidr_block        = cidrsubnet(aws_vpc.main.cidr_block, 2, each.key)
}
```

LECTURE

Intro to Modules

What are Modules?

A module is a container for related resources that are used together.

They are the primary way to package and reuse configurations.

Improved Organization

Break large configurations into logical units to easily reuse over and over again.

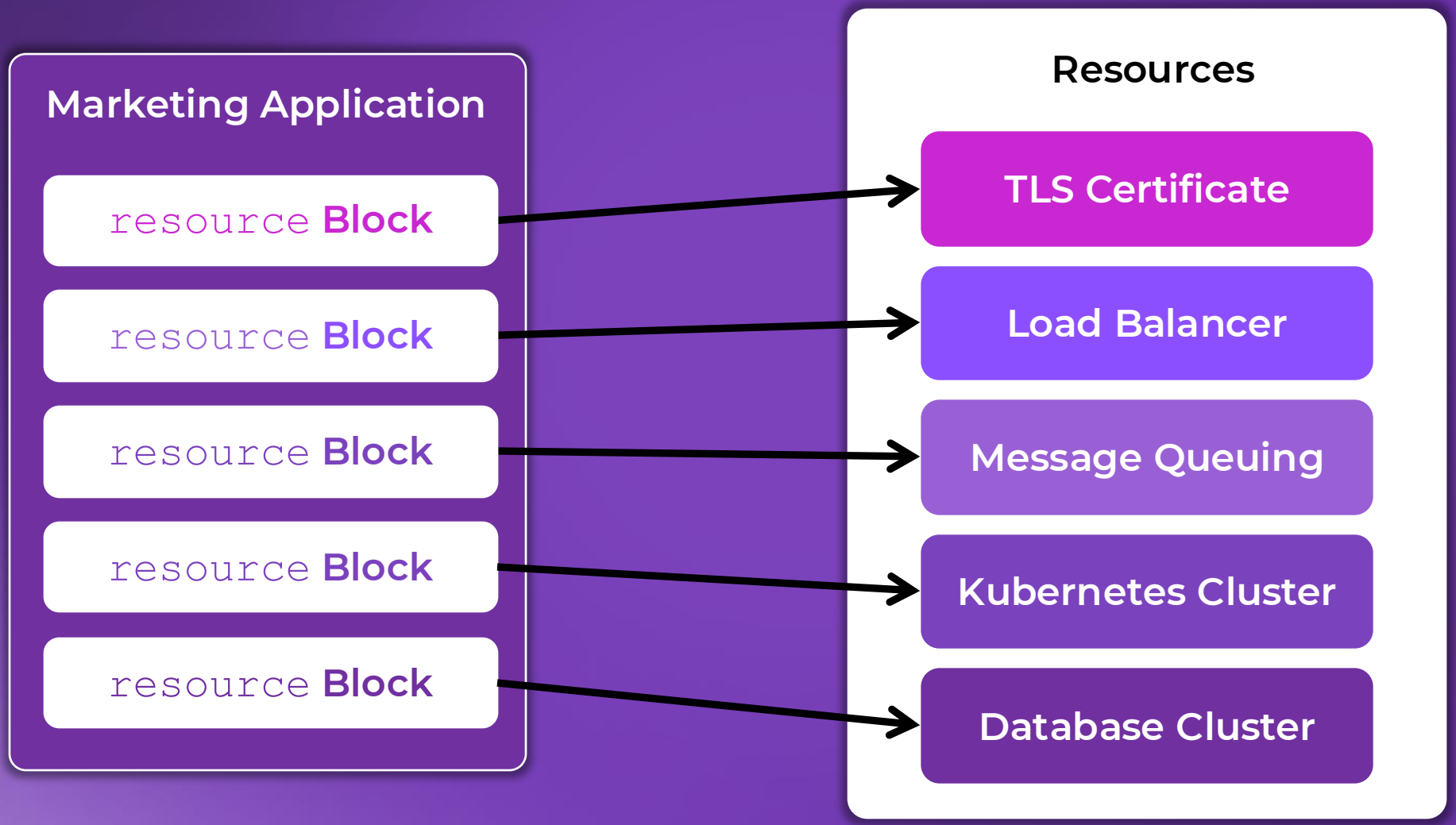
Easier Collaboration

Teams can share and maintain standardized building blocks or publish approved modules for organizational use.

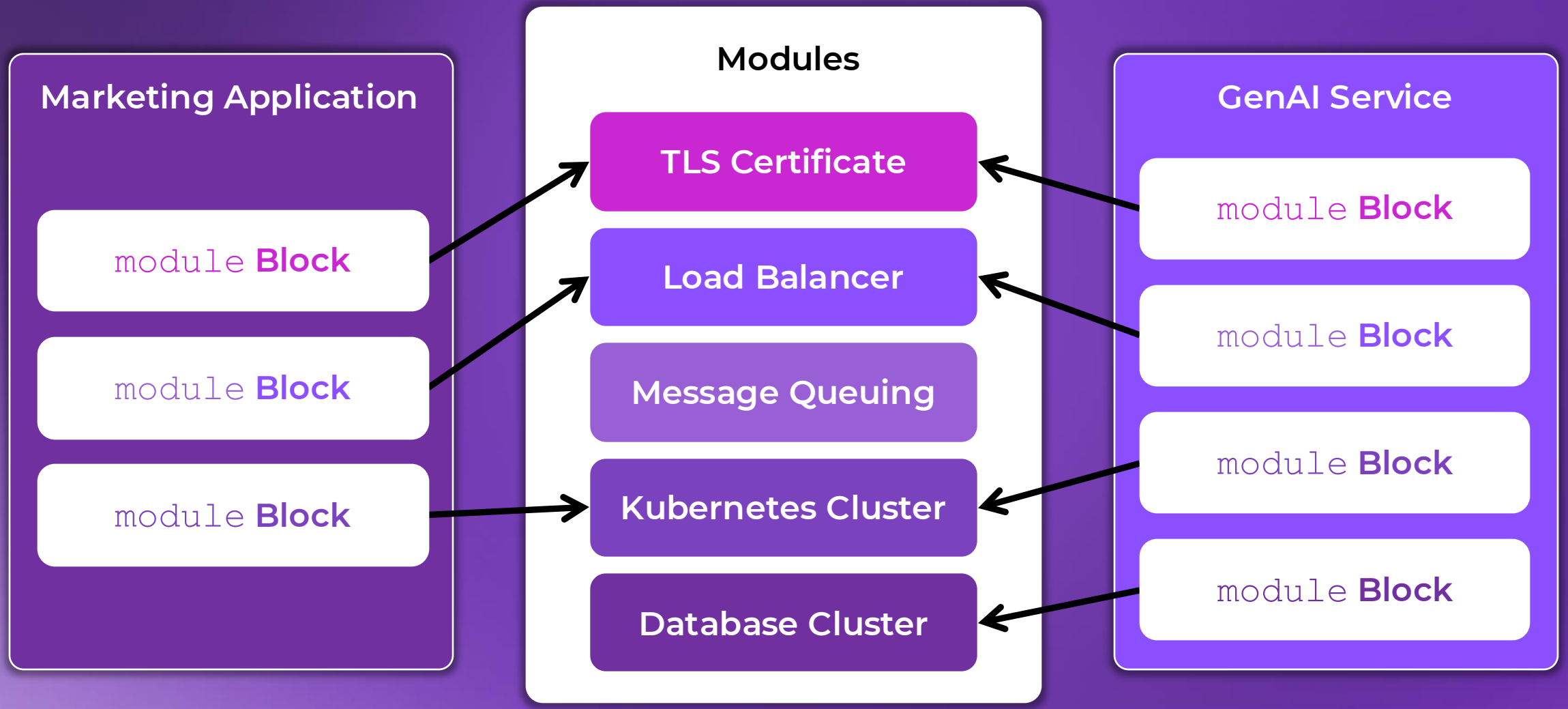
Consistent Patterns

Enforce naming conventions, security controls, and best practices across deployments.

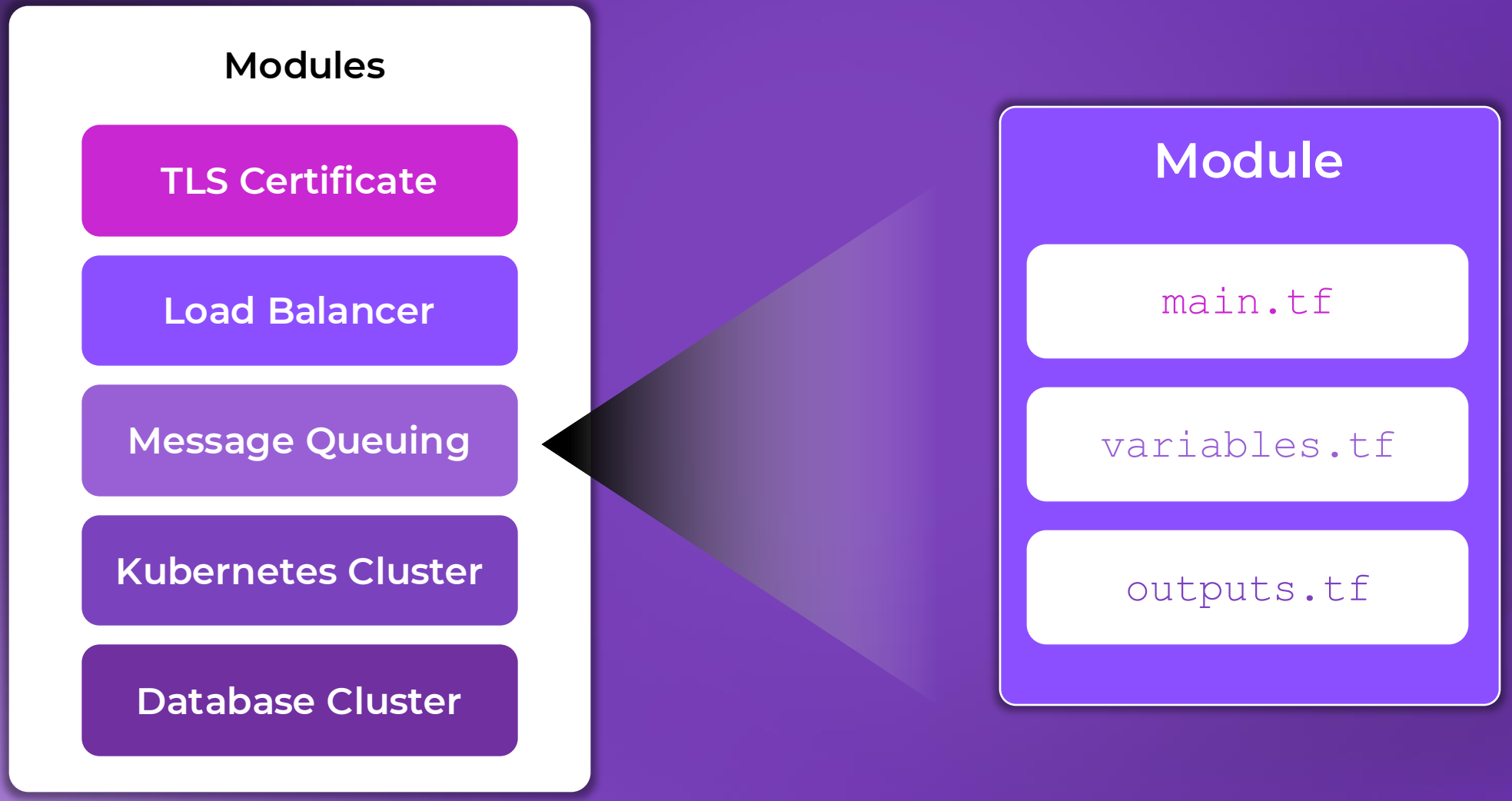
Terraform Building Blocks



Terraform Building Blocks



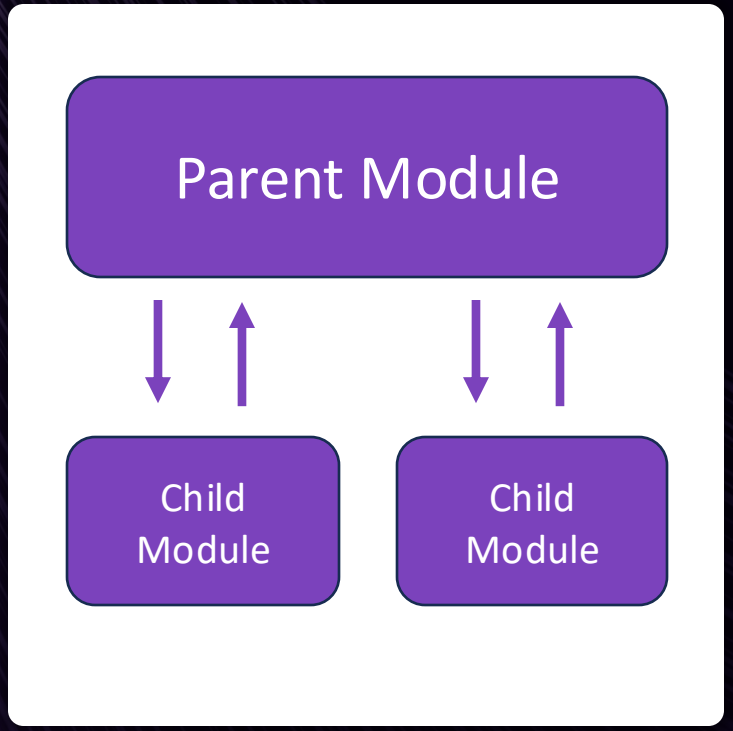
Terraform Building Blocks



Module Block

Parent Module vs. Child Module

- A parent (calling) module references and configures other modules, while a child module is the reusable infrastructure component being called.
- Modules can be retrieved from a registry, a repo, or stored locally.



Parent/Root/Calling Module

```
1  module "vpc" {
2    source = "terraform-aws-modules/vpc/aws"
3
4    name = "my-vpc"
5    cidr = "10.0.0.0/16"
6
7    azs             = data.aws_availability_zone.east.name_suffix
8    private_subnets = var.private_subnets
9    public_subnets  = var.public_subnets
10
11   enable_nat_gateway = true
12   enable_vpn_gateway = true
13
14   tags = {
15     CreatedBy   = "terraform"
16     Environment = "prod"
17   }
18 }
```

Module Blocks Call Child Modules
Defines that this module is the calling module of the child module

Must Define Where the Module Lives
Path to local directory or remote module source to download from

Variables Defined in Root Module
Values must be passed to the module to be used

Very Simple Child Module

```hcl
variable "cidr_block" {
  type    = string
  default = "10.0.0.0/16"
}

variable "name" {
  type    = string
  default = ""
}

resource "aws_vpc" "vpc" {
  name       = var.name
  cidr_block = var.cidr_block

  tags = {
    Name = var.name
  }
}

output "vpc_id" {
  value = aws_vpc.vpc.id
}
```

Modules Should Look Familiar

Modules are just regular Terraform configurations that are variablized

Must Define One or More Resources

The child module is where the resource blocks are defined

Child Modules Have Outputs

Outputs allow you to pass information from one module to another

Root Module

```hcl
variable "cidr_block" {
  type    = string
  default = "192.168.0.0/16"
}

module "prod_vpc" {
  source = "terraform-aws-modules/vpc/aws"

  name = "my-vpc"
  cidr = var.cidr_block
}

output "vpc_id" {
  module.prod_vpc.vpc_id
}
```

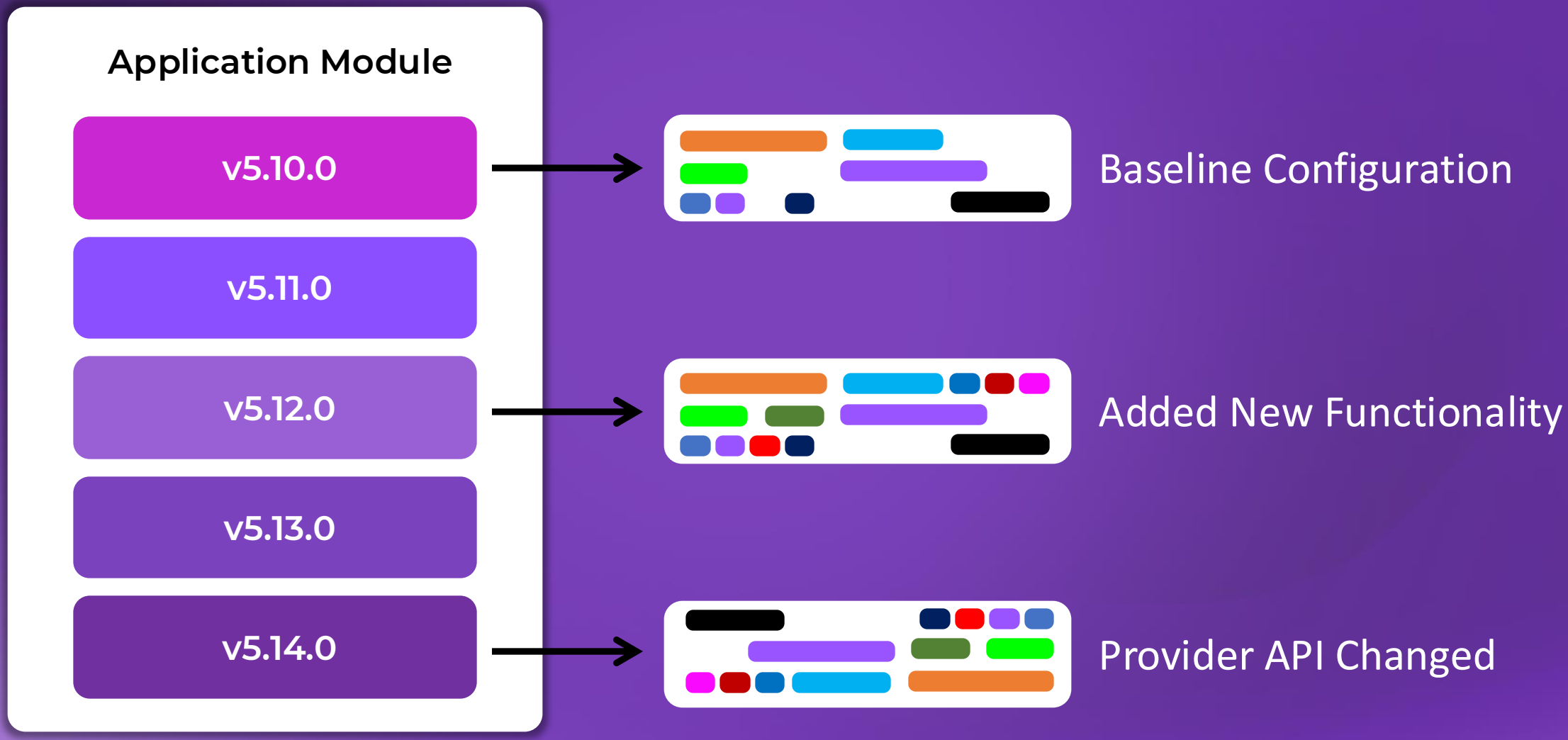
Values we want to pass to our module

Child Module

```hcl
variable "cidr" {
  type    = string
  default = "10.0.0.0/16"
}

variable "name" {
  type    = string
  default = ""
}

resource "aws_vpc" "vpc" {
  name       = var.name
  cidr_block = var.cidr

  tags = {
    Name = var.name
  }
}

output "vpc_id" {
  value = aws_vpc.vpc.id
}
```

Export values to be accessed by calling the module

Module Versioning



Pinning to a Specific Module Version

```
1  variable "cidr_block" {
2    type    = string
3    default = "192.168.0.0/16"
4  }
5
6  module "prod_vpc" {
7    source = "terraform-aws-modules/vpc/aws"
8    version = "5.12.0"
9
10   name = "my-vpc"
11   cidr = var.cidr_block
12 }
13
14 output "vpc_id" {
15   module.prod_vpc.vpc_id
16 }
```

SUMMARY

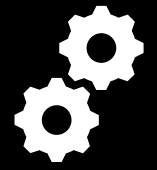
HCL is a declarative language that is easy to read and write, and is Terraform's primary interface.

A single Terraform file can have one or many blocks to define your infrastructure

HCL uses blocks for data, resources, etc., and these blocks can be nested.

Use the '.tf' file extension so Terraform and related tools can recognize and apply configurations properly.

Single-line and block comments are used in HCL.



Use linters, extensions, and built-in tools like 'terraform fmt' to enhance code quality.