Course Introduction

Wayne Hoggett
Azure Training Architect

@WayneHoggett

LESSON BREAKDOWN

Course Introduction

Introducing Infrastructure as Code

Terraform Workflow

Scenario Introduction

Wayne Hoggett
Azure Training Architect

Introducing Infrastructure as Code

Chocolate Cake

Ingredients

- Flour
- Cocoa powder
- Sugar

Method

1. Pre-heat oven.
2. Mix ingredients in bowl.

Script

1. Create a resource group.
2. Create a virtual network.
3. Create a public IP address.
4. Create a network interface with the public IP address.
5. Create a virtual machine with the network interface.

Introducing Infrastructure as Code

Script

1. Create a resource group.
2. Create a virtual network.
3. Create a public IP address.
4. Create a network interface with the public IP address.
5. Create a virtual machine with the network interface.

Infrastructure as Code Advantages

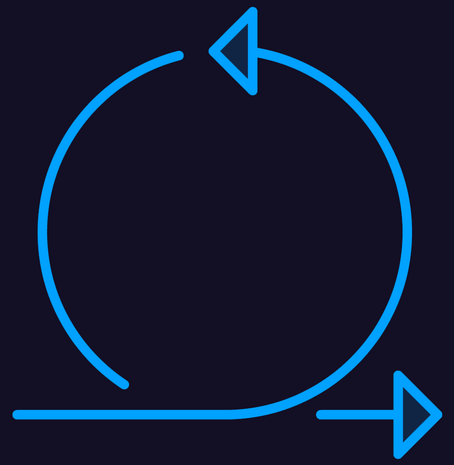
- Idempotency through state
- Automatic dependencies
- Version control with Git
- Reusable code through modules

Terraform Workflow

Scenario Introduction

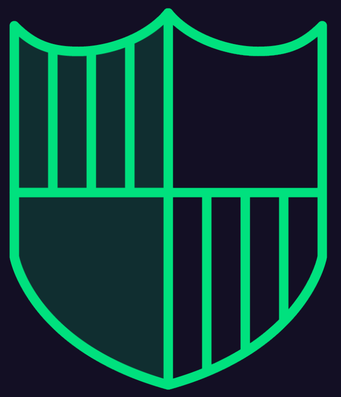
River City
ARTIFICIAL INTELLIGENCE

- Specialized machine-learning company focused on speech-to-text and voice analysis
- Already using Amazon Web Services and are interested in adopting Azure data and AI services



Agility

- Continuous delivery
- Reusable code
- Version control
- Remote state



Security

- Secure secrets
- Secure modules
- Extensible pipeline

Summary

- Terraform has several advantages, including:
  - Idempotency
  - Modules
  - Automatic dependencies
  - Version control integration
- You can deploy infrastructure using Terraform by:
  - Authoring your configuration files
  - Initializing the environment including providers and state
  - Planning and deploying your infrastructure

Setting Up Your Infrastructure-as-Code Workstation

Wayne Hoggett
Azure Training Architect

@WayneHoggett

LESSON BREAKDOWN

Setting Up Your Infrastructure-as-Code Workstation

Your Infrastructure-as-Code Toolkit

Demo: Setting Up Your Infrastructure-as-Code Workstation

Wayne Hoggett
Azure Training Architect

Your Infrastructure-as-Code Toolkit

Terraform
Validate, initialize, plan, and deploy your infrastructure.

Git
Version control and collaborate on your code.

Azure CLI
Log in to Azure and create credentials for Terraform Cloud.

Visual Studio Code
Author your configuration.

Demo

Setting Up Your Infrastructure-as-Code Workstation

- Set up your Infrastructure-as-Code workstation *as code* so you have a consistent development environment.
- Chocolatey can be used to automate the installation of packages on Windows.
- Manually install Visual Studio Code extensions.

Summary

- To use Terraform to deploy to Azure using Infrastructure as Code, you'll need a few tools, including:
  - Terraform
  - Git
  - Azure CLI
  - Visual Studio Code
- Use Chocolatey to automate the setup of your Infrastructure as Code workstation.

Deploying to Azure Using Terraform

Wayne Hoggett
Azure Training Architect

@WayneHoggett

LESSON BREAKDOWN

Deploying to Azure Using Terraform

Configuration Building Blocks

Demo: Deploying to Azure Using Terraform

Wayne Hoggett
Azure Training Architect

```
terraform {
  required_version = "~> 1.3.8"
  cloud {
    organization = "RiverCityAI"
    workspaces {
      name = "mlstorage"
    }
  }
  required_providers {
    azurerm = {
      "source" = "hashicorp/azurerm"
      version  = "3.43.0"
    }
  }
}

provider "azurerm" {
  features {}
  skip_provider_registration = true
}

resource "azurerm_resource_group" "mlstoragerg" {
  name     = "rg-mlstorage-dev-001"
  location = "Australia East"
}
```

- Terraform configuration block, including:
- Required Terraform version
- Backend or cloud configuration
- Required providers
- Provider configuration block
- Resource block

Demo

Deploying to Azure Using Terraform

1. Author our configuration.
2. Initialize the working directory.
3. Import existing resources.
4. Create an execution plan.
5. Execute our plan.

Summary

- Terraform configuration is declared using blocks.
- State can be stored remotely by declaring a `backend` or using Terraform Cloud.
- Import existing resources into you state using `terraform import`.

Managing Terraform State

Wayne Hoggett
Azure Training Architect

@WayneHoggett

LESSON BREAKDOWN

Managing Terraform State

The Good News

The Bad News

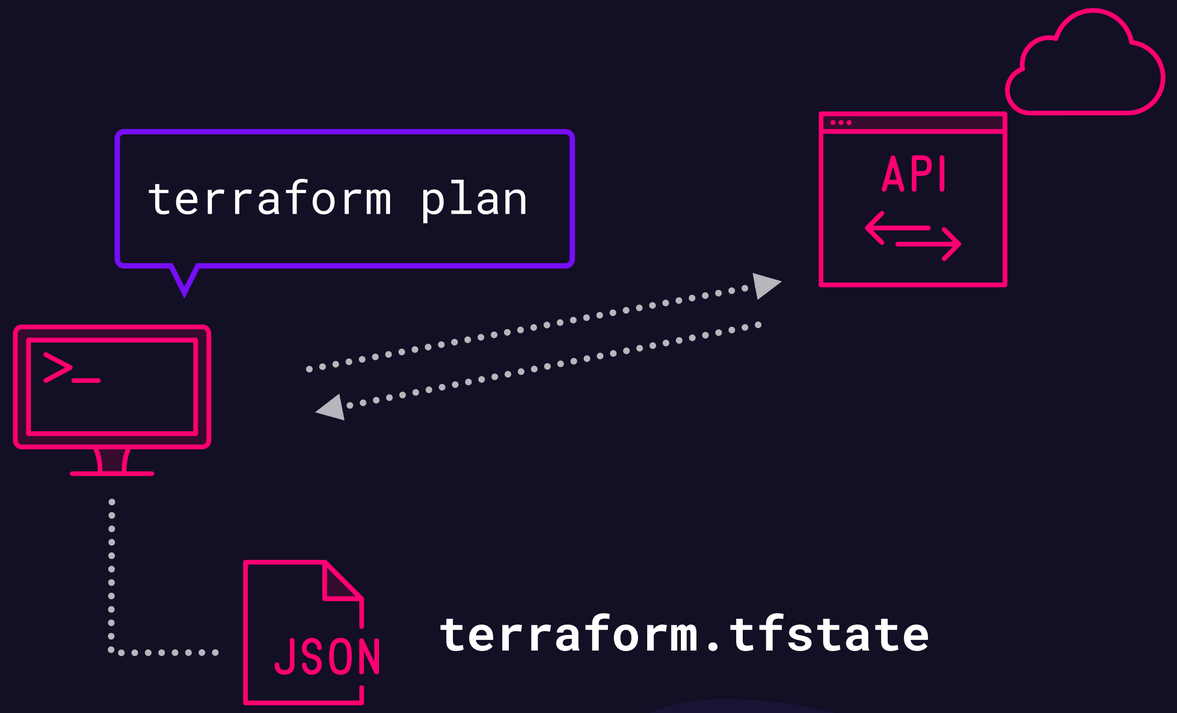
Demo: Migrate Terraform State to Terraform Cloud

Wayne Hoggett
Azure Training Architect

The Good News

Terraform state:

- Maps configuration to resources in the real world
- Enables dependency mapping
- Improves performance



The Bad News

My configuration

`terraform.tfstate`

The Bad News

Azure Storage

`terraform.tfstate`

The Bad News

Remote state enables:

- Encryption of the state file
- Collaboration

Demo

Migrate Terraform State to Terraform Cloud

1. Create and view local state.
2. Create a Terraform Cloud workspace.
3. Migrate state to Terraform Cloud.
4. Delete the local state file.

Summary

- Terraform state is used for configuration and dependency mapping.
- Terraform state improves performance.
- Sensitive data in your state files can be protected by using a backend.
- Configuring a backend provides remote state, which enables collaboration.

Providing Continuous Delivery

Wayne Hoggett
Azure Training Architect

@WayneHoggett

LESSON BREAKDOWN

Providing Continuous Delivery

Introducing Version Control

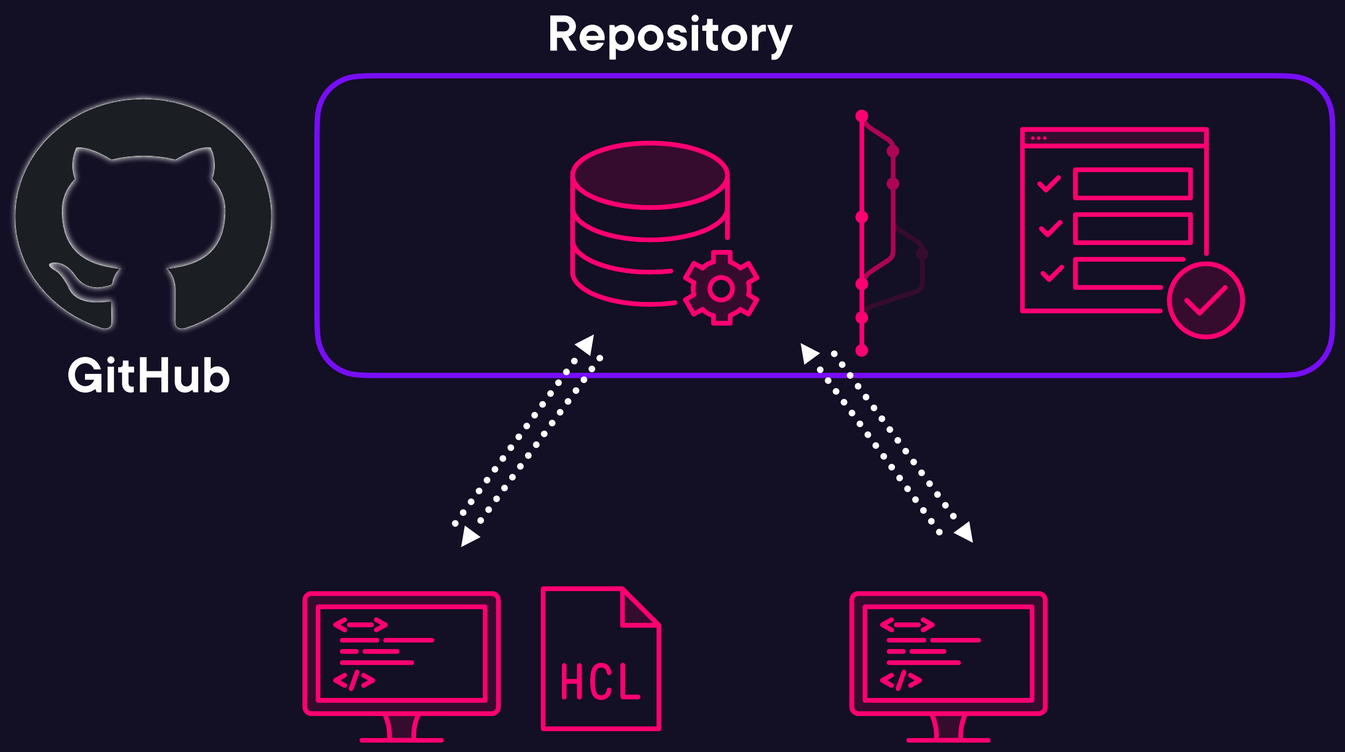
Introducing Continuous Delivery

Continuous Delivery Workflow

Demo: Continuous Delivery with GitHub Actions

Wayne Hoggett
Azure Training Architect

Introducing Version Control



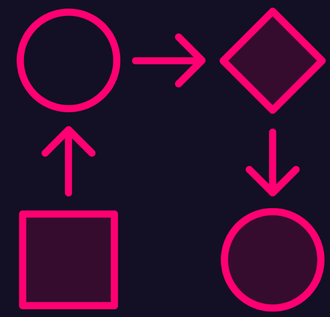
Use a Version Control System to:

- Centrally store your Infrastructure as Code in a repository.
- Store a version history for your infrastructure.
- Review infrastructure changes, and protect production workloads with *branches*.

Introducing Continuous Delivery

Continuous delivery enables automatic deployment based on code changes.

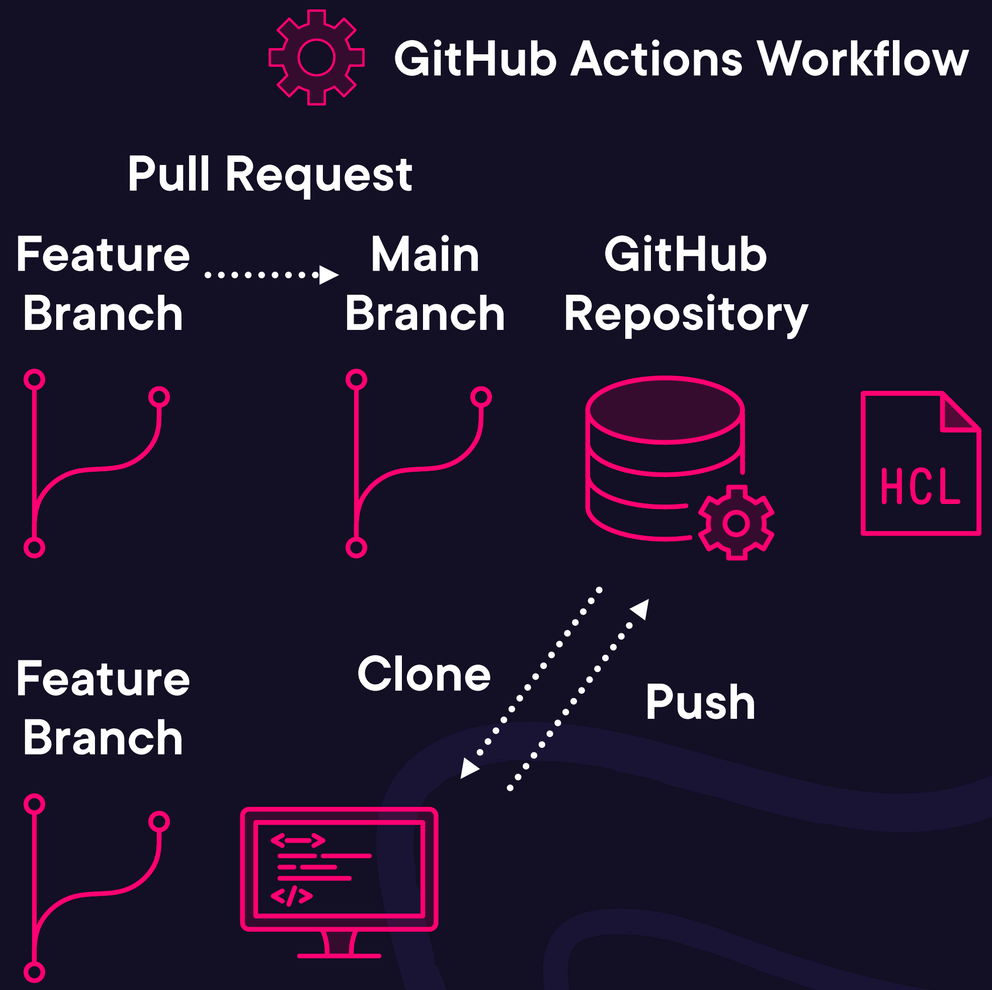
Terraform Cloud provides basic Version Control Service (VCS) integration.



GitHub Actions provides change approvals and extensible workflows.

Continuous Delivery Workflow

- The *main* branch represents the production infrastructure.
- A *feature* branch represents your work in progress.
- A *pull request* is used to merge code between branches.



Demo

Continuous Delivery with GitHub Actions

1. Create a GitHub repository.
2. Configure continuous delivery.
3. Configure branch protection.
4. Configure Terraform Cloud.
5. Use the Git workflow to author, stage, commit, and push our configuration.
6. Create and approve the pull request.

Summary

- A Version Control System allows you to centrally store your code, keep a version history, and have your code reviewed by others.
- Git *branches* are used to represent environments and work in progress.
- A p*ull request* is used to merge code between branches.

Effectively Managing Terraform Code

Wayne Hoggett
Azure Training Architect

@WayneHoggett

LESSON BREAKDOWN

Effectively Managing Terraform Code

Leveraging Terraform Variables

Exploring Terraform Modules

Demo: Create a Terraform Module and Publish It to a Private Registry

Wayne Hoggett
Azure Training Architect

Effectively Managing Terraform Code

Leveraging Terraform Variables

```
resource "azurerm_resource_group" "rg" {
  name     = "rg-modeldata-prod-001"
  location = "Australia East"
}

resource "azurerm_storage_account" "storageaccount" {
  name                     = "stmodeldataprod001"
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}
```

Leveraging Terraform Variables

```
variable "storage_account_replication_type" {
  type = string
  default = "LRS"
}

resource "azurerm_resource_group" "rg" {
  name     = "rg-modeldata-prod-001"
  location = "Australia East"
}

resource "azurerm_storage_account" "storageaccount" {
  name                     = "stmodeldataprod001"
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
  account_replication_type = var.storage_account_replication_type
}
```

- Define *input variables* for commonly changed resource arguments.
- Use the values stored in variables with the *var* keyword.

Effectively Managing Terraform Code

Leveraging Terraform Variables

```
variable "storage_account_replication_type" {
  type    = string
  default = "LRS"
}

locals {
  workload_name = "modeldata"
  environment   = "prod"
  instance = "001"
}

resource "azurerm_resource_group" "rg" {
  name     =  "rg-${local.workload_name}-${local.workload_name}-$
{local.instance}"
  location = "Australia East"
}

resource "azurerm_storage_account" "storageaccount" {
  name                     = "st{local.workload_name}$
{local.workload_name}${local.instance}"
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
  account_replication_type = var.storage_account_replication_type
}
```

- Use *locals* to avoid repeating the same values multiple times in a configuration.
- Use local values with the *local* keyword.

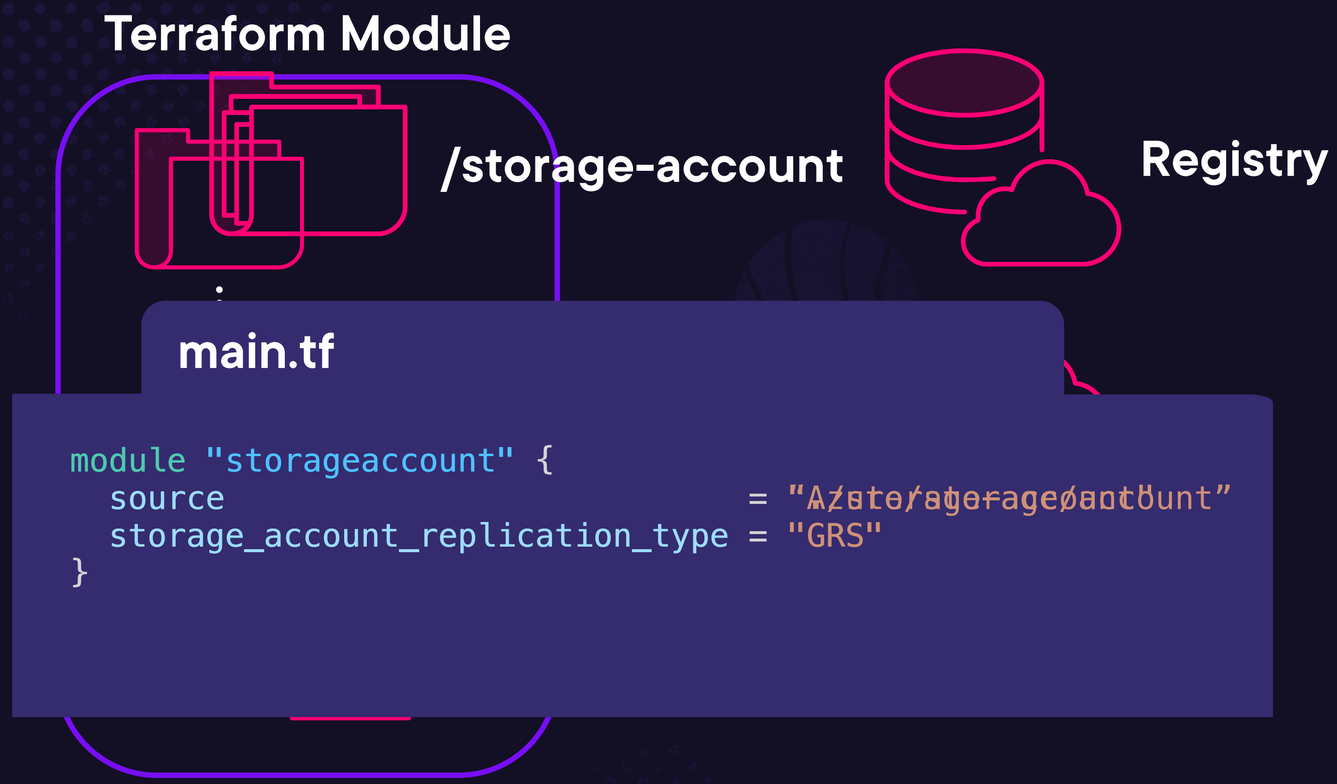
Effectively Managing Terraform Code

Leveraging Terraform Variables

```
variable "storage_account_replication_type" {
  type    = string
  default = "LRS"
}

locals {
  workload_name = "modeldata"
  environment   = "prod"
  instance = "001"
}

resource "azurerm_resource_group" "rg" {
  name     =  "rg-${local.workload_name}-${local.workload_name}-$
{local.instance}"
  location = "Australia East"
}

resource "azurerm_storage_account" "storageaccount" {
  name                     = "st{local.workload_name}$
{local.workload_name}${local.instance}"
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
  account_replication_type = var.storage_account_replication_type
}

output "storage_account_primary_blob_host" {
  value = azurerm_storage_account.storageaccount.primary_blob_host
}
```

- Use *output variables* to export data from a configuration.

Exploring Terraform Modules



- A collection of Terraform configuration files in a single directory is a *module*.
- Modules can be deployed directly.
- Modules can also be reused by other modules.
- Modules can stored in a *registry* to share the module with others.

Demo

Create a Terraform Module and Publish It to a Private Registry

1. Create a GitHub repository for our module.
2. Author a reusable Terraform module.
3. Upload our module to our private registry in Terraform Cloud.
4. Create a Terraform configuration that uses the module from the private registry.

Summary

- Use Terraform variables to make code reusable and reduce repetition in code.
- Use Terraform modules to create reusable resources that enable consistency between resources.
- The Public Terraform Registry provides a reusable Terraform modules.
- Terraform Cloud provides a private registry you use to share modules internally.

Course Conclusion

Wayne Hoggett
Azure Training Architect

@WayneHoggett

Course Conclusion

Our Journey

River City
ARTIFICIAL INTELLIGENCE

Security

- Secure secrets
- Extensible pipeline
- Secure modules

Agility

- Remote state
- Version control
- Continuous delivery
- Reusable code

Thank You

Wayne Hoggett

Azure Training Architect

@WayneHoggett