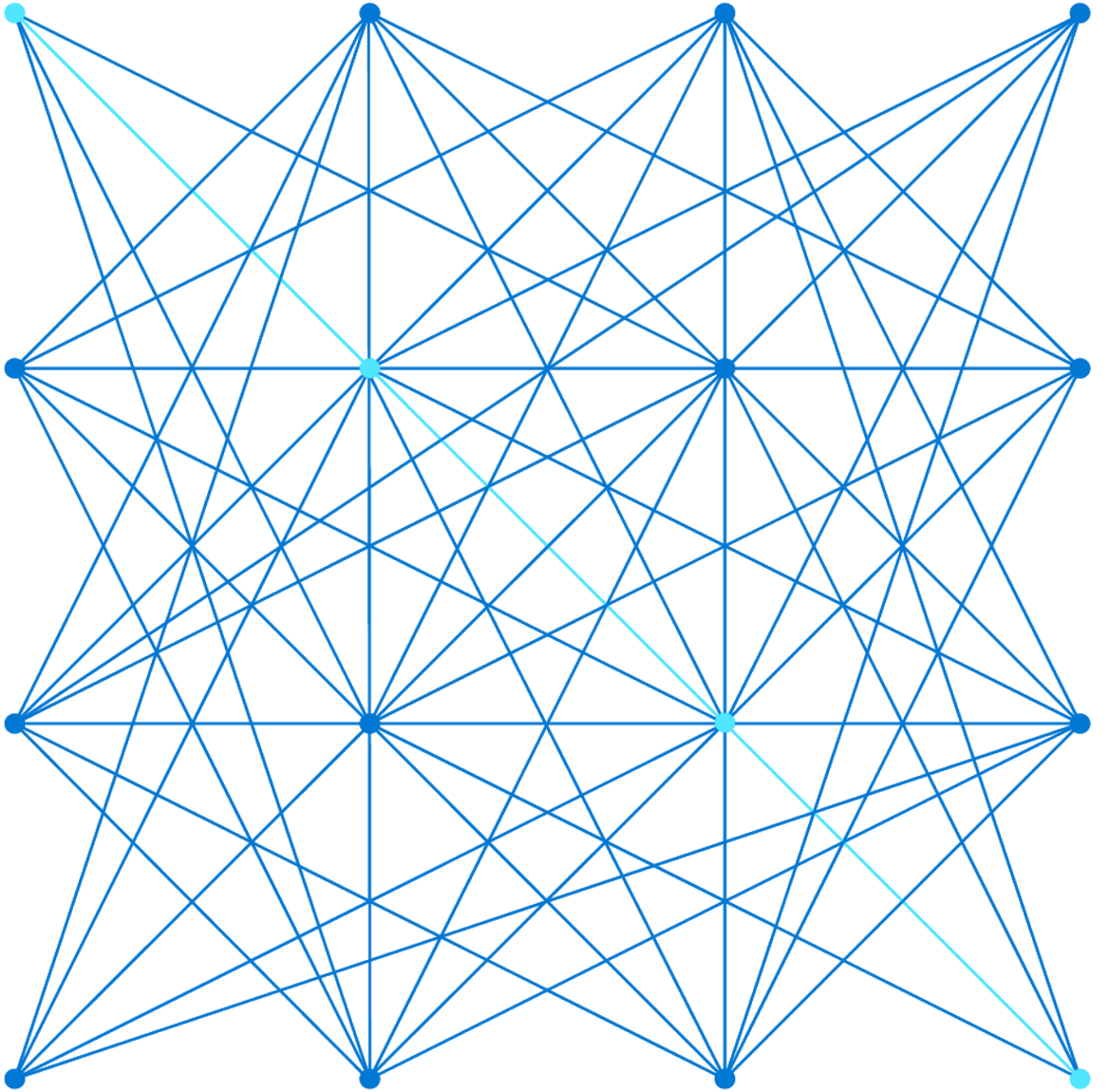
Microsoft Azure

# Terraform

Damien Aicheh

Cloud Solution Architect

damienaicheh@microsoft.com



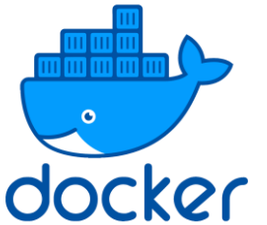
# Format of this training

- Training for Terraform **beginners**
- Alternate between **theory and practice**
- **6 blocs of 4 hours** to focus on Terraform
- For the time of the training, you’re blocked from customer and other internal calls
- There are **no stupid questions** so don't hesitate to ask
- At the end of this training, you will have a good understanding of the way to start a Terraform project

# Prerequisites for each participant

- An Azure account with **contributor** rights
- Visual Studio Code with Terraform plugin:
  - https://marketplace.visualstudio.com/items?itemName=hashicorp.terraform
- In your local machine:
  - **Terraform**
  - **Azure CLI**
  - **Docker (optional)**
- Access to GitHub Actions / Ability to create repositories

GitHub Actions



# HashiCorp Terraform

# HashiCorp

- HashiCorp is based in San Francisco and was founded in **2012**
- HashiCorp provides a suite of **open-source tools** intended to support development and deployment of large-scale service-oriented software installations.
- Each tool is aimed at **specific stages in the life cycle of a software** application, with a focus on automation

# HashiCorp current products

## INFRASTRUCTURE

**Terraform**
Infrastructure as code

**Packer**
Machine images

## SECURITY



**Vault**
Identity-based security

**Boundary**
Secure remote access

## NETWORKING

**Consul**
Multi-cloud service networking

## APPLICATIONS

**Nomad**
Workload orchestration

**Waypoint**
App deployment workflows

**Vagrant**
Environment workflows

# Terraform

- HashiCorp Terraform enables you to safely and predictably **create, change, and improve infrastructure.**
- **Codifies APIs** into **declarative configuration** files that can be shared among team members, treated as code, edited, reviewed, and versioned.
- **Abstraction layer** over common public/private cloud provisioning/automation languages
- Declarative Config Files are written in **HCL** (HashiCorp Configuration Language)
- **Single binary** without complex installation
- **Cross platform**, can be run on Windows, Linux, or MacOS
- Written in Go with **plugin-oriented architecture**

# Terraform

- What Terraform is **NOT** write once, deploy anywhere
- **Providers** are the libraries for the different vendors
- Providers have **different settings**
- Terraform enables deployment in Azure, AWS and Google using **different** providers, configured with **different settings**
- Terraform use the **same langage** but **not** the **same resource**:
  Azure: Storage Account
  AWS: S3
  GCP: Cloud Storage


# Terraform

- Terraform enables **management of multi cloud** environments using a single toolset and syntax
- Broad **community**, very **responsive**
- **Frequent releases:** approximately every two weeks
- **Extensible**
- Open source / **no vendor** lock-in
- Microsoft is heavily investing in making **most Azure services supported**
- Microsoft **add Terraform support** on there products, like Azure Dev Cli (azd)

# Terraform: Caveats

- Very **new, or advanced Azure functionality** may not be natively available in Terraform Provider
- Workarounds do exist: the **azurerm_template_deployment** resource can be used to **wrap a native ARM Template**, but this has its own limitations (it doesn't support resource deletions)
- **Unencrypted secrets** in state file
- You have to manage the **state** of your platform

# Licencing

- On August 10, 2023, HashiCorp announced a change of license for its products
- Terraform being open source under the **MPL v2 license**, it was to move under a non-open source **BSL v1.1 license**
- Starting from the version **1.6**
- At this point it impacts only the company **building products based on** the Terraform technology
- Companies that deploy resources using Terraform today can **continue** to do it **for free**
- Initiative with Opentofu: https://github.com/opentofu/manifesto

# Hands On Lab!

# 2 options:

- Install everything on your machine

- Use DevContainers inside Visual Studio Code and Docker:

https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-containers

https://github.com/damienaicheh/terraform_tooling

# Check your configuration

Just run in your favorite terminal:

For Terraform:


For Azure CLI:

# Set up your Azure account

```
# Login to Azure
az login

# Show your account
az account show

# Select your subscription
az account set --subscription <your-subscription-id>
```

# HCL Language

# HCL (HashiCorp Configuration Language)

**One language** and Cross Platform

**Human** and machine **readable**

***.TF** files, UTF-8 encoded

Blocks containing key/value pairs

HCL **is not JSON**
- Less nesting
- Less brackets

Supports inline comments
#, //, /*  */

**Case sensitive**

**Naming** convention: **snack case**

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

// Comments here...
resource "azurerm_storage_account" "example" {
  name                     = "storageaccountname"
  resource_group_name      = azurerm_resource_group.example.name
  location                 = azurerm_resource_group.example.location
  account_tier             = "Standard"
  account_replication_type = "GRS"

  tags = {
    environment = "staging"
  }
}
```

# ARM Template vs Terraform HCL

Terraform HCL is about 50-70 % more compact than ARM.

**ARM JSON**

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

**Terraform HCL**

```
name = "${var.PilotServerName}-vm"
```

# ARM Template vs Terraform HCL

| ARM Template | Terraform |
| --- | --- |
| JSON | HCL |
| Parameters | Variables |
| Variables | Local variables |
| Resources | Resources |
| Functions | Functions |
| Nested templates | Modules |
| Explicit dependency | Automatic dependency |
| Refer by reference or resourceId | Refer by resource or data source |

# Terraform variables, locals and outputs

| | Variables | Locals | Outputs |
|---|---|---|---|
| What | Parameters | Local variables | Results |
| Example | Environment name | Resource group name based on combination of variables, operations, prefix and suffix | Resource Identifier, Resource Url |
| How to reference it? | var.<name> | local.<name> | module.<module name>.<output name> |

# How to declare variables, locals and outputs

```
variable "resource_group_name_suffix" {
  type        = string
  description = "The resource group name suffix"
}
```

```
locals {
    resource_name_suffix = format("%s-%s", var.environment, var.owner)
}
```

```
output "app_service_url" {
    value = azurerm_app_service.this.default_site_hostname
    description = "The URL of the App Service"
}
```

# Optional arguments: Description

Always a string

```
variable "resource_group_name_suffix" {
  type        = string
  description = "The resource group name suffix"
}
```

```
output "app_service_url" {
  value       = azurerm_app_service.this.azurerm_app_service
  description = "The URL of the App Service"
}
```

# Optional arguments: sensitive

```
variable "my_service_password" {
  type        = string
  description = "My service password"
  sensitive = true
}
```

Boolean (true/false)

Suppressing Values in CLI Output

**Secrets must be stored in Key Vault directly**

**Secrets are stored unencrypted in the state**

```
output "db_password" {
  value       = aws_db_instance.db.password
  description = "The password for logging in to the database."
  sensitive   = true
}
```

# Optional arguments: depends_on

Explicit dependency

**Does not exist for variables**

```
output "default_site_hostname" {
  value       = azurerm_app_service.this.default_site_hostname
  description = "The default site hostname."

  depends_on = [
    azurerm_app_service.this
  ]
}
```

# Optional arguments: default

Default value means that the variable become optional

**Does not exist for outputs**

```
variable "resource_group_name_suffix" {
  type        = string
  default     = "01"
  description = "The resource group name suffix"
}
```

# Optional arguments: type

Types is optional but recommended to restrict the values like:

- string
- number
- bool
- list<Type>
- set(<Type>)
- map(<Type>)
- object({<ATTR NAME> = <TYPE>, ... })
- tuple([<TYPE>, ...])

**Does not exist for outputs**

# Optional arguments: type

| Category | Type | Description | Example: Declare | Example: Access Value |
|---|---|---|---|---|
| **Any** | Any | Any type / no constraint | | var.myVar |
| **Primitive Type** | string | Double quoted, with common escape sequences: \n, \t, \\, \"<br><br>Multi-line strings „heredoc" format. | "\"multi\"\nline"<br><<EOT<br>multiline<br>heredoc string<br>EOF | var.myVar |
| | number | whole number, as well as fractions | 12<br>3.3475 | var.myVar |
| | bool | true / false | | var.myVar |
| **Complex Types** | list(<TYPE>) | sequence of values, 0-based index | ["us-west-1a", "us-west-1c"] | var.myVar[0] |
| | set(<TYPE>) | Collection of unique values | | var.myVar[0] |
| | tuple([<TYPE>, ...]) | Similar to list, but each element may have different types | ["us-west-1a", 12] | var.myVar[0] |
| | map(<TYPE>) | group of values identified by named labels | {name = "Mabel", city = "Munich"} | var.myVar.name<br>var.myVar["name"] |
| | object({<ATTR NAME> = <TYPE>, ... }) | Similar to map, but attributes may have different types | {name = "Mabel", age = 52} | var.myVar.name<br>var.myVar["name"] |

# Optional arguments: validation

Advanced validation rules like regex or conditions

**Does not exist for outputs**

```
variable "resource_group_name_suffix" {
  type        = string
  default     = "01"
  description = "The resource group name suffix"
  validation {
    condition     = can(regex("[0-9]{2}", var.resource_group_name_suffix))
    error_message = "The resource group name suffix value must be two digits."
  }
}
```

# Optional arguments: nullable

Boolean (true/false), whether null is allowed or not for the variable, default is true

**Does not exist for outputs**

```
variable "environment" {
  type     = string
  nullable = false
}
```

# Terraform Expressions

| Category | Syntax | Description |
|---|---|---|
| **Operators** | 1. !, - (multiplication by -1)<br>2. *, /, %<br>3. +, - (subtraction)<br>4. >, >=, <, <=<br>5. ==, !=<br>6. &&<br>7. \|\| | Ordered operators<br>== requires same type / no implicit conversion<br><br>In general: Terraform is case sensitive<br>=> “a“ != “A“<br>=> Also case for function names matters! |
| **Conditional expression** | condition ? true_val : false_val<br>e.g.<br>var.a != "" ? var.a : "default-a" | Implicit conversion, if results don’t have same type<br>=> string + number results in string |
| **For expression** | [for k, v in var.map : length(k) + length(v)] | Variable for key k is optional.<br>In case of map and object, it contains the key.<br>For list, set and tuple, it contains the index number. |

# Terraform Functions

| Category | Functions - see official docs for full list |
|---|---|
| **Numeric** | • Functions with 1 or 2 param: **abs, ceil, floor, log, parseint, pow, signum**<br>• **max** and **min**: numeric functions expecting 1..n arguments<br>  • Syntax: max(1, 3, 4)<br>  • Or expand list using „...“: max([1, 3, 4]...)<br>• **sum**: collection function, only expecting list: sum([1, 3, 4]) |
| **String** | **chomp, format, formatlist, join, lower, regex, regexall, replace, split, strrev, substr, title, trim, upper** |
| **String / Collection** | **length** |
| **Collections** | • Checks: **alltrue, anytrue, contains**,<br>• Returning collections: **chunklist, coalescelist, compact, concat, distinct, flatten, keys, range, reverse, slice, values**<br>• Returning primitive types: **coalesce, index, lookup**,<br>• Returning maps or objects: **merge, transpose, zipmap**<br>• Returning sets: **setintersection, setproduct, setsubstract, setunion** |
| **Encoding** | **base64gzip, csvdecode, jsondecode, jsonencode, textdecodebase64, textencodebase64, urlencode, yamldecode, yamlencode** |
| **Date and time** | **formatdate, timeadd, timestamp** |
| **Hash and Crypto** | **md5, sha256, sha512, rsadecrypt, uuid** |
| **Type Conversion** | **tobool, tolist, tomap, tonumber, toset, tostring, try** |

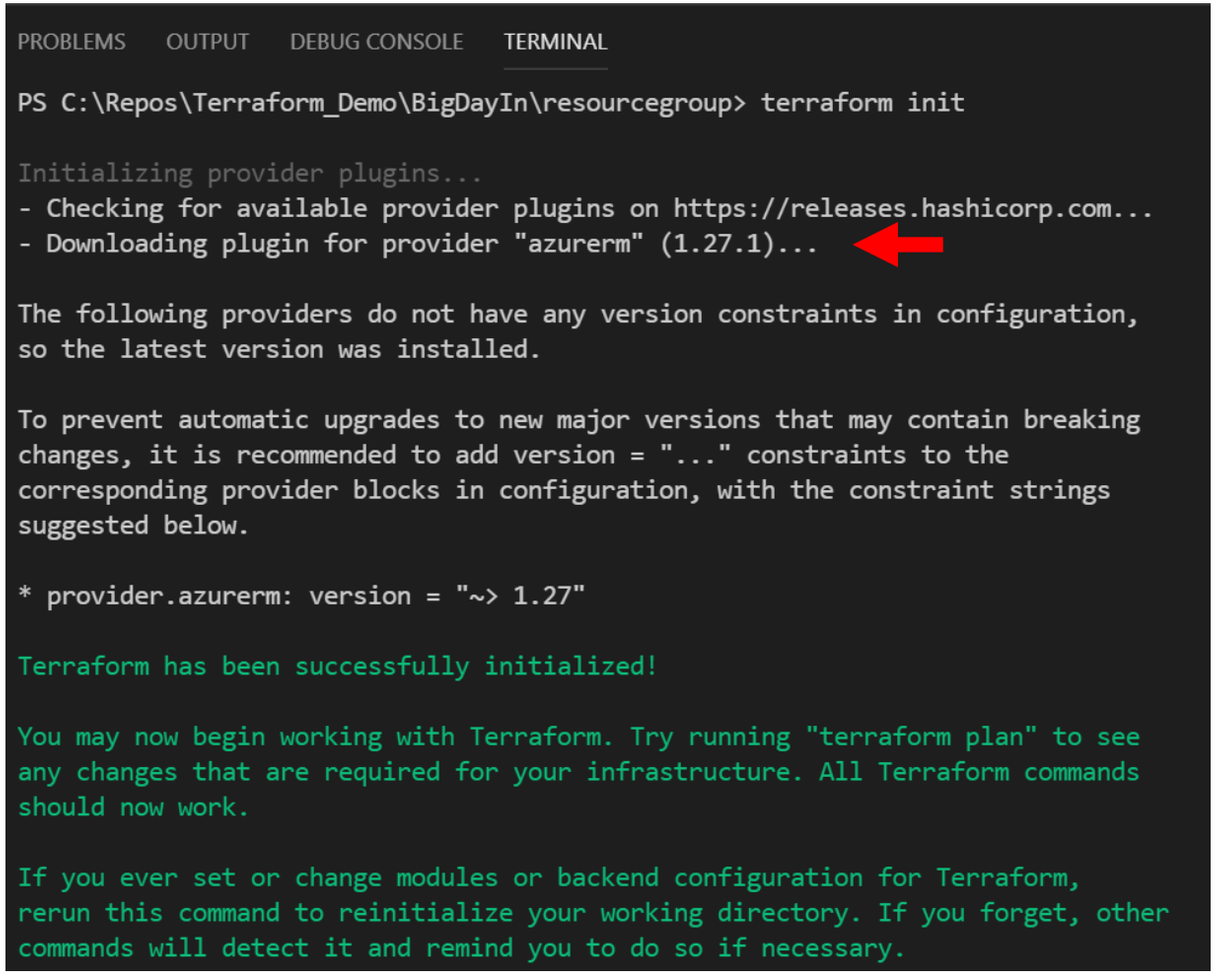
# Terraform Workflow and commands

# Terraform Workflow: init

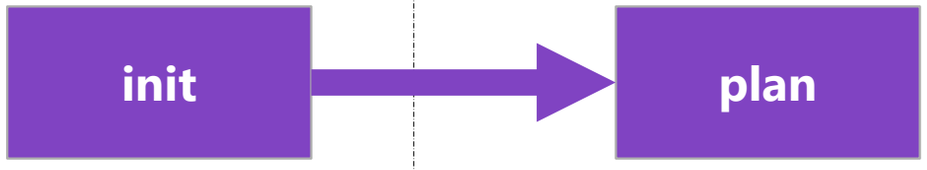

- **Initializing** working directory
- **Download** provider version
- **Analyzing** modules
- **Backend** initialization

# Terraform Workflow: init

- Before running *terraform init*, **connect to Azure** using az CLI to obtain an **authorization token**, and select the **subscription** you wish to deploy to.
- Run the **init** command in the directory that contains the ***.tf*** files to be deployed.
- It determines which **providers** to download based on the .tf files in the current directory.
- Provider files are stored within the current directory in a hidden directory "***.terraform***"



# Terraform Workflow: plan




- **Initializing** working directory
- **Download** provider version
- **Analyzing** modules
- **Backend** initialization

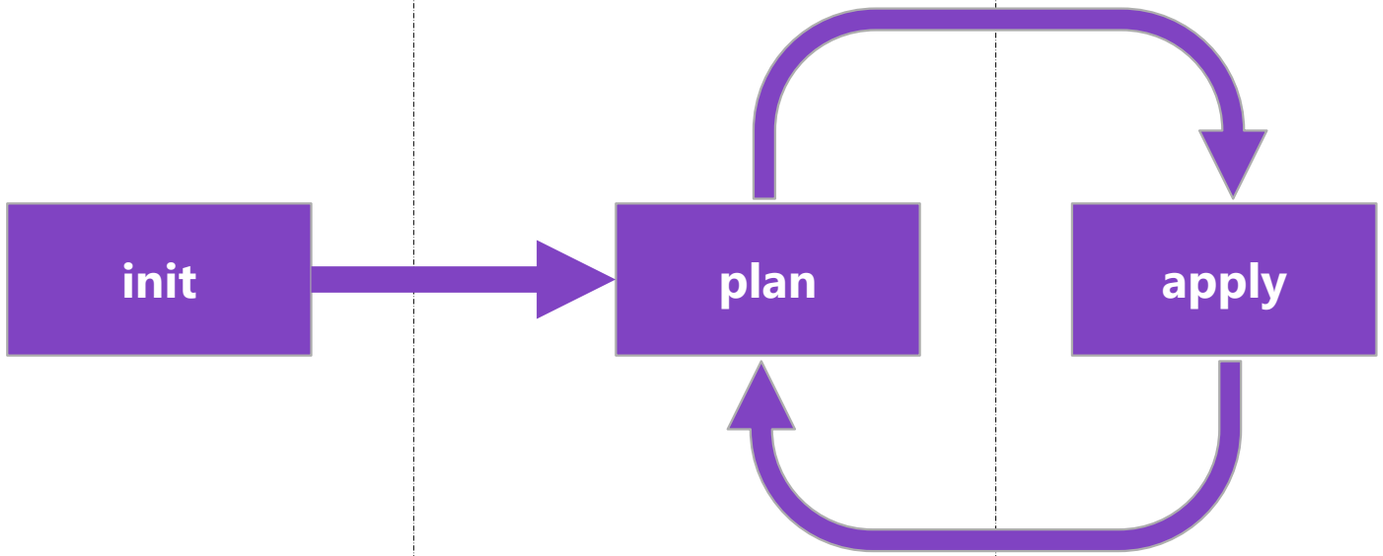
- **Compares** code changes to current state
- Creates a list of instruction to **migrate the state to the desired changes**
- Creates a **plan file** with the list of instruction (no execution)

# Terraform Workflow: plan

- **Plan** command provide a **preview of the actions**
- It's an opportunity to **sanity check** the deployment.
- Resource **additions** will be indicated by a + symbol, and highlighted in green
- Resource **modifications** will be indicated by a ~ symbol and highlighted in orange
- Resource **deletions** will be indicated by a – symbol and highlighted in red
- **Colours** only shown **if supported by terminal**
- Where required, specify the ***–var*** and ***–var-file*** parameters to pass in variable values to the **plan** command.

```
PS C:\Repos\Terraform_Demo\BigDayIn\resourcegroup> terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.


------------------------------------------------------------------------

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  + azurerm_resource_group.rg
      id:       <computed>
      location: "australiaeast"
      name:     "DEVOPStestresourcegroup"
      tags.%:   <computed>


Plan: 1 to add, 0 to change, 0 to destroy.

------------------------------------------------------------------------

Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.
```

# Terraform Workflow: apply




- **Initializing** working directory
- **Download** provider version
- **Analyzing** modules
- **Backend** initialization

- **Compares** code changes to current state
- Creates a list of instruction to **migrate the state to the desired changes**
- Creates a **plan file** with the list of instruction (no execution)

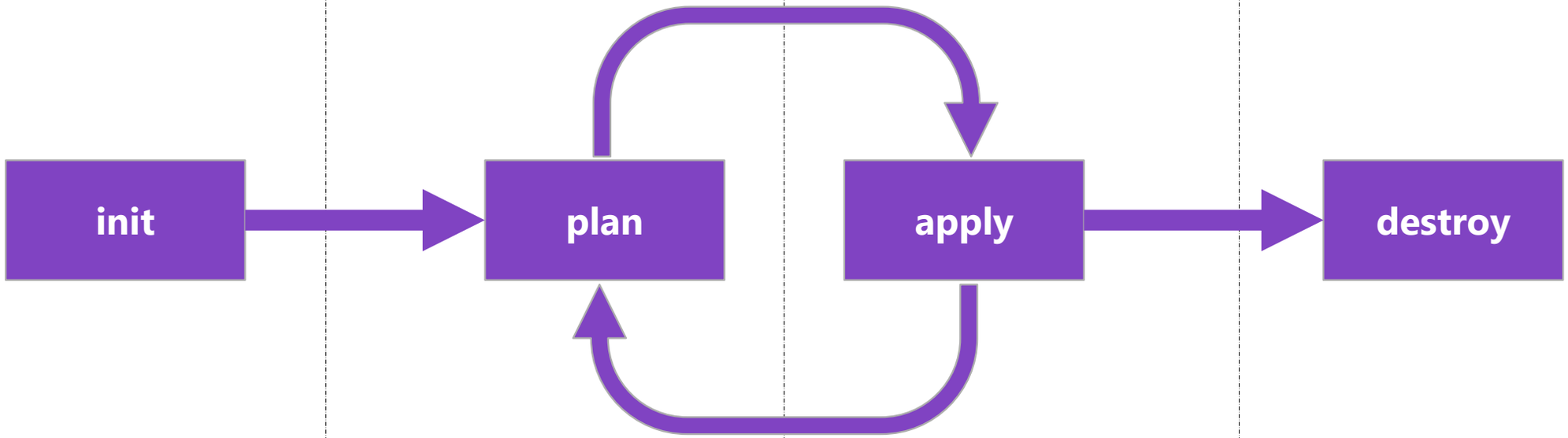
- Actual infrastructure: **Applies** the set of instruction created in the plan file

# Terraform Workflow: apply

- The **apply** command is used to execute the terraform deployment and add, change and delete resources.
- For automated deployments the ***-auto-approve*** parameter is used to skip the confirmation step.
- Where required, specify the ***–var*** and ***–var-file*** parameters to pass in variable values to the **apply** command.
- This command **can take some time** to run for large deployments. Continuous output will be provided to assist in monitoring progress.

```
PS C:\Repos\Terraform_Demo\BigDayIn\resourcegroup> terraform apply

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  + azurerm_resource_group.rg
      id:       <computed>
      location: "australiaeast"
      name:     "DEVOPStestresourcegroup"
      tags.%:   <computed>


Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

azurerm_resource_group.rg: Creating...
  location: "" => "australiaeast"
  name:     "" => "DEVOPStestresourcegroup"
  tags.%:   "" => "<computed>"
azurerm_resource_group.rg: Creation complete after 3s (ID: /subscriptions/85b

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

# Terraform Workflow: destroy




- **Initializing** working directory
- **Download** provider version
- **Analyzing** modules
- **Backend** initialization

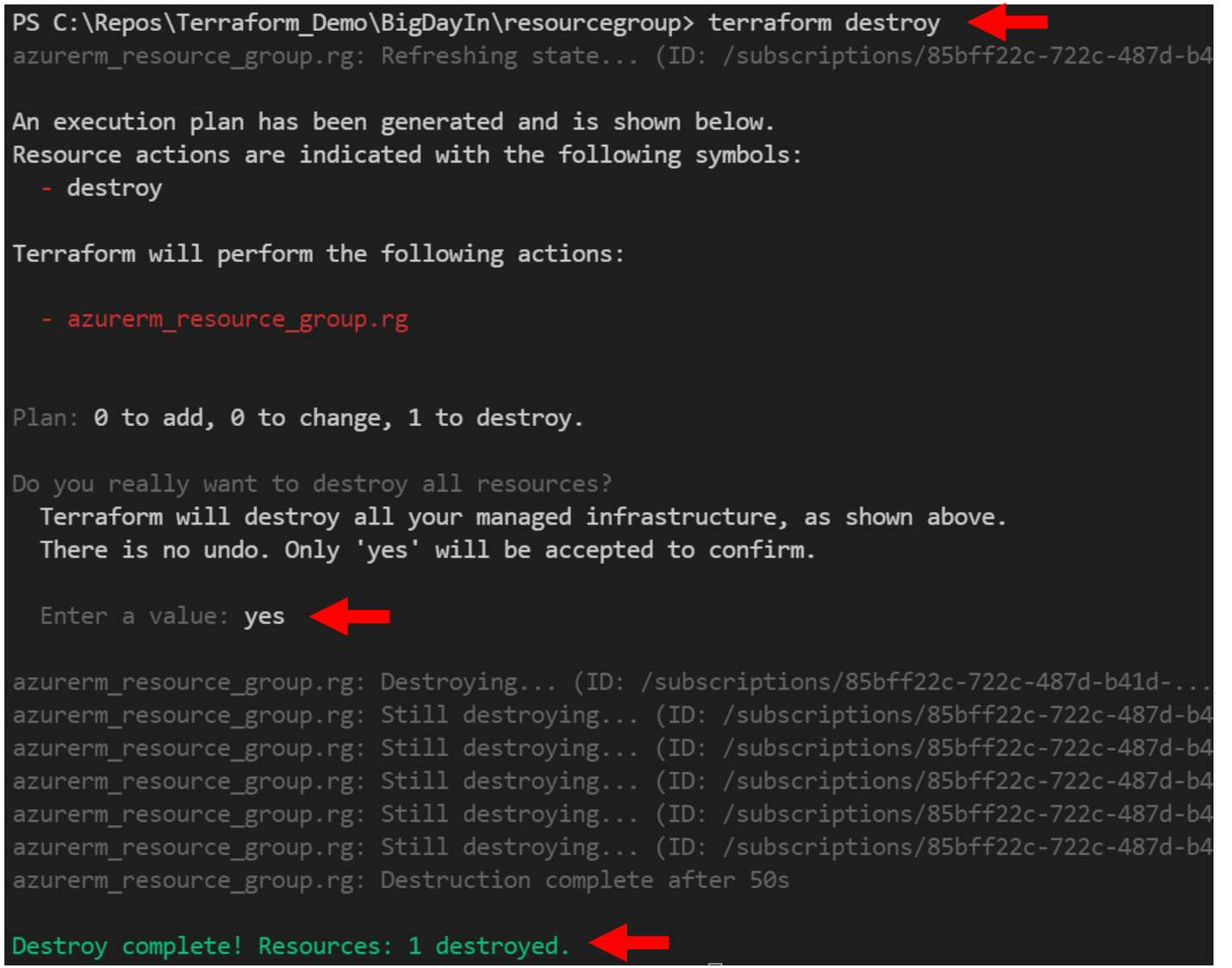
- **Compares** code changes to current state
- Creates a list of instruction to **migrate the state to the desired changes**
- Creates a **plan file** with the list of instruction (no execution)

- Actual infrastructure: **Applies** the set of instruction created in the plan file

- Actual infrastructure: **Remove** all configured resources

# Terraform Workflow: destroy

- The **destroy** command will remove all resources specified in the deployment .tf files.
- When executed, a **plan is printed** to show the resources that will be destroyed, and you are prompted for confirmation.
- For automated deployments the ***-auto-approve*** parameter is used to skip the confirmation step.
- Where required, specify the ***–var*** and ***–var-file*** parameters to pass in variable values to the **destroy** command.
- This command **can take some time** to run for large deployments. Continuous output will be provided to assist in monitoring progress.



# Terraform commands

| Command | Description |
| --- | --- |
| **terraform init** | Initialize the context of the project (providers, modules, backends…) |
| **terraform fmt -recursive** | Format the tf files in a standard manner |
| **terraform validate** | Validate the syntax of the tf files **BUT** there is no guarantee that the deployment will succeed |
| **terraform plan** | Define the potential creation, updates, deletion based on the current tfstate of the architecture |
| **terraform apply** | Apply the changes based on the plan.out if provided |
| **terraform destroy** | Destroy all the resources defined in the tfstate |

# Hands On Lab!

# HCL: HashiCorp Configuration Language

- Play with variable file provided (main.tf)
- Use **terraform init** to initialize the project
- To test the code just run:

  **terraform plan**

  **terraform apply**

- The instructions are on the next slide

# HCL: HashiCorp Configuration Language

- Output the **application_name** with only the 3 first characters
- Output the **letters** array in **one single string** with **dashes** to separate each letter
- Create a variable **string** called **domain** and when you output it make sure it's **lowercase**
- Return the **node count** in a dedicated output
- Return the **node size** in a dedicated output
- Output a resource group name which will be a **concatenation** of the official abbreviation "rg" and the **application name** and **domain** formatted previously. Dashes must separate each part.
  E.g. rg-<application name>-<domain>

# Resources & Data Sources

# Resources

- Most **important element** in the Terraform language.
- Each resource block **describes** one or more **infrastructure** objects
- Syntax:

```
resource "[Resource-Type]" "Resource-Name" {
  [Optional parameters]
}
```

- https://www.terraform.io/language/resources

# Resources

- Syntax example for an Azure resource group:

```
resource "azurerm_resource_group" "this" {
  name     = var.resource_group_name
  location = var.location
}
```

- Syntax to access to values:

```
azurerm_resource_group.this.location
```

# Data Sources

- Use information defined **outside of Terraform**, for instance Azure
- Use information defined by **another** separate Terraform **configuration** (state)
- Syntax:

```
data "[Data-Source-Type]" "Data-Name" {
  [Optional parameters]
}
```

- https://www.terraform.io/language/data-sources

# Data Sources

- Syntax example for data source to declare the access the configuration of the AzureRM provider:

```
data "azurerm_client_config" "current" {}
```

- Syntax to access the values, for instance the subscription id:

```
data.azurerm_client_config.current.subscription_id
```

# Terraform State

# Terraform State: Overview

- Terraform **stores the state** about your managed infrastructure and configuration **in a file**
- The state is used to **map real world resources to the configuration**
- Terraform **uses this state to create plans** and make changes to your infrastructure
- **Sensitive information** is **stored unencrypted** within state
- **terraform show** can be used to display managed resources in the state

# Terraform State: Local vs. Remote

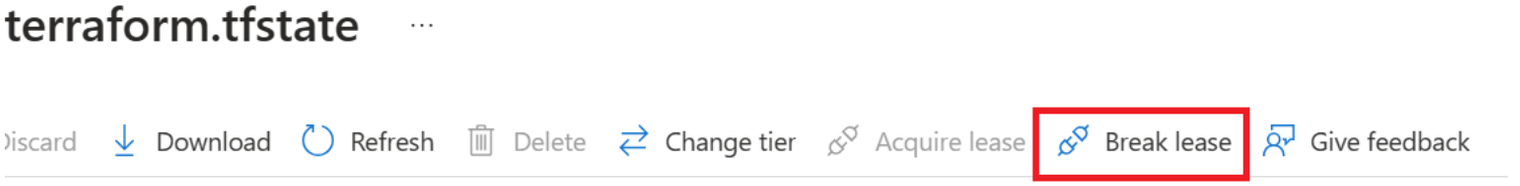
- The state is stored by default in a **local file** named **terraform.tfstate**
- It's configured using backend block in the provider resource
- Local State – for **non-prod tests only**
- Remote State – Should be **stored remotely** (e.g. blob) to enable **security and collaboration**
- Before each deployment, Terraform must be able to **obtain a lock on the state file**.
- Once the **deployment** is **complete**, the **lock is released**.
- **Azure Storage Accounts** support **file locking** which prevents multiple people from simultaneously executing a Terraform deployment, **security**, **version control** and **disaster recovery** capabilities for your state files.

# Terraform State: Remote

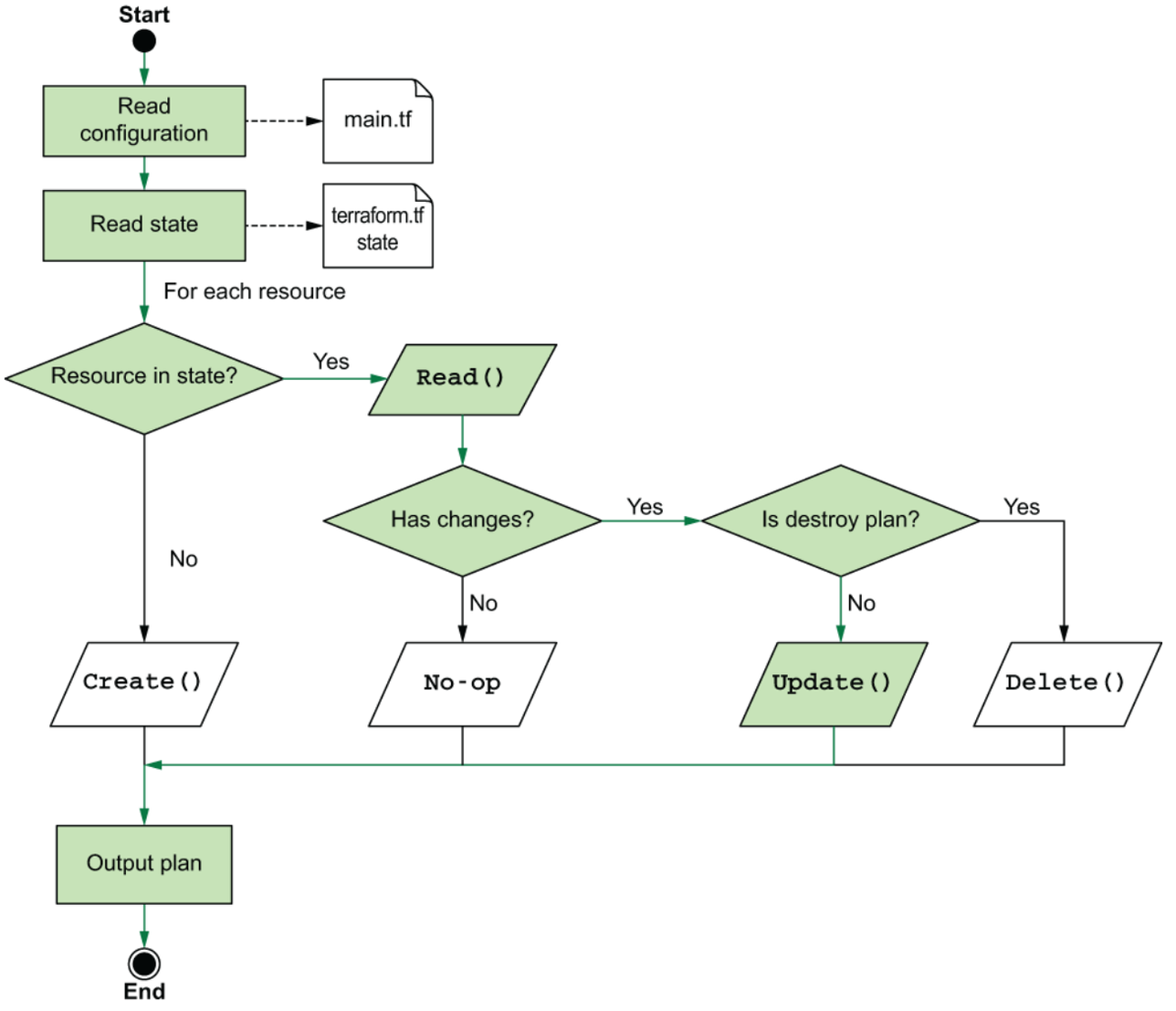
- Terraform has his own cloud: **Terraform Cloud Remote State Management**

- **Azure Storage Accounts** support **file locking** which prevents multiple people from simultaneously executing a Terraform deployment, **security**, **version control** and **disaster recovery** capabilities for your state files.

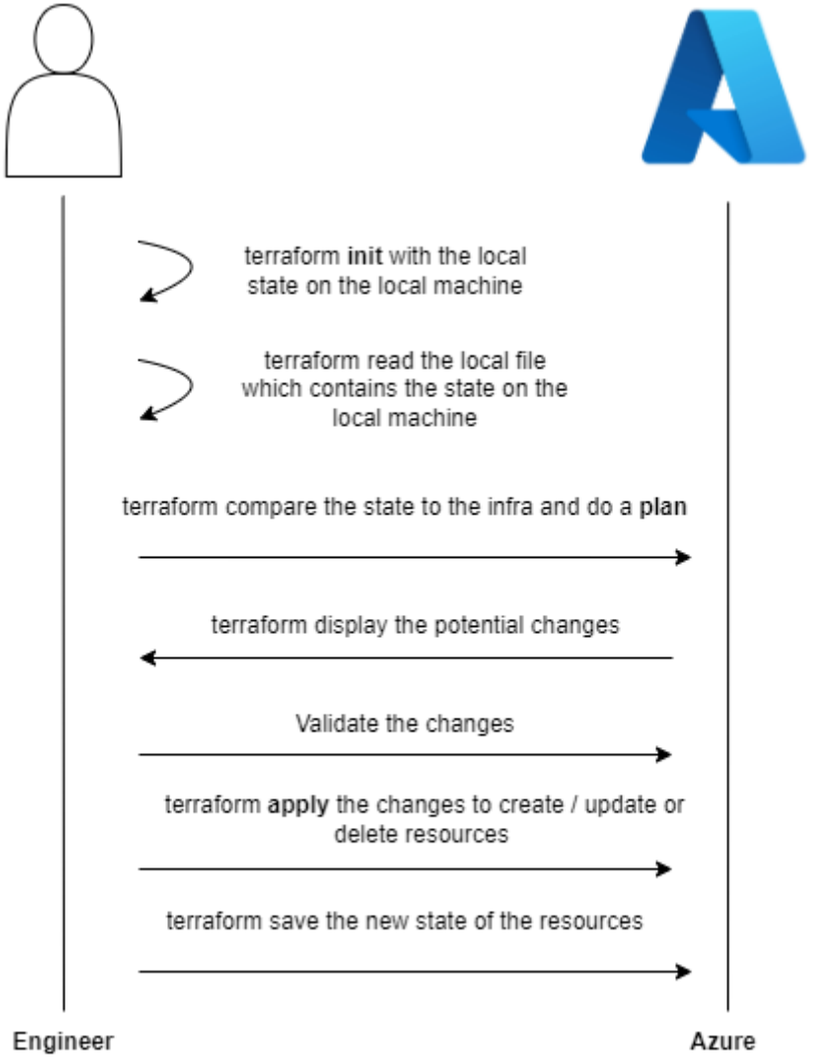
- If a state is **lock**, you will have to connect to Azure Storage Account and click “Break lease”



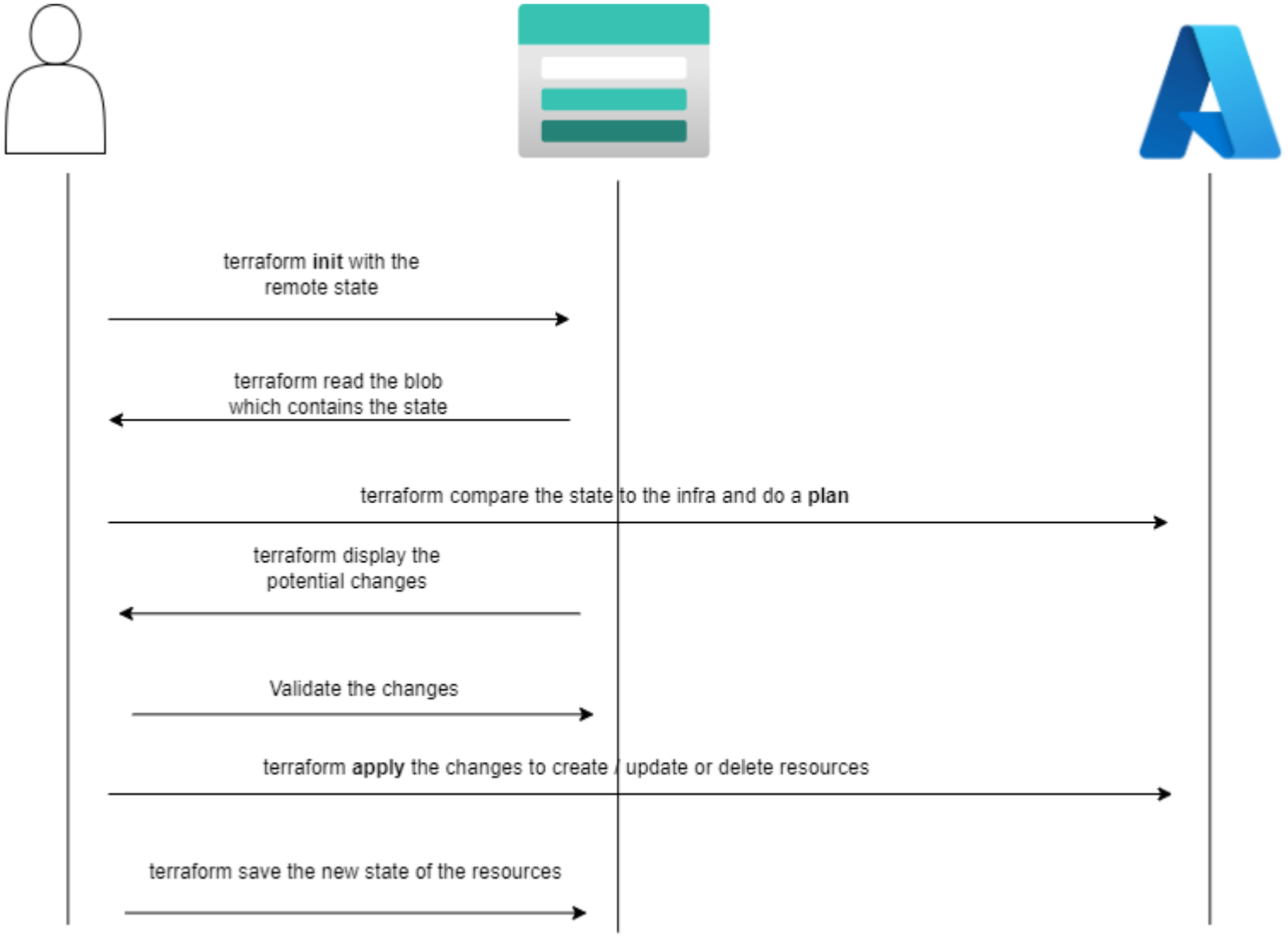
# How the plan is generated



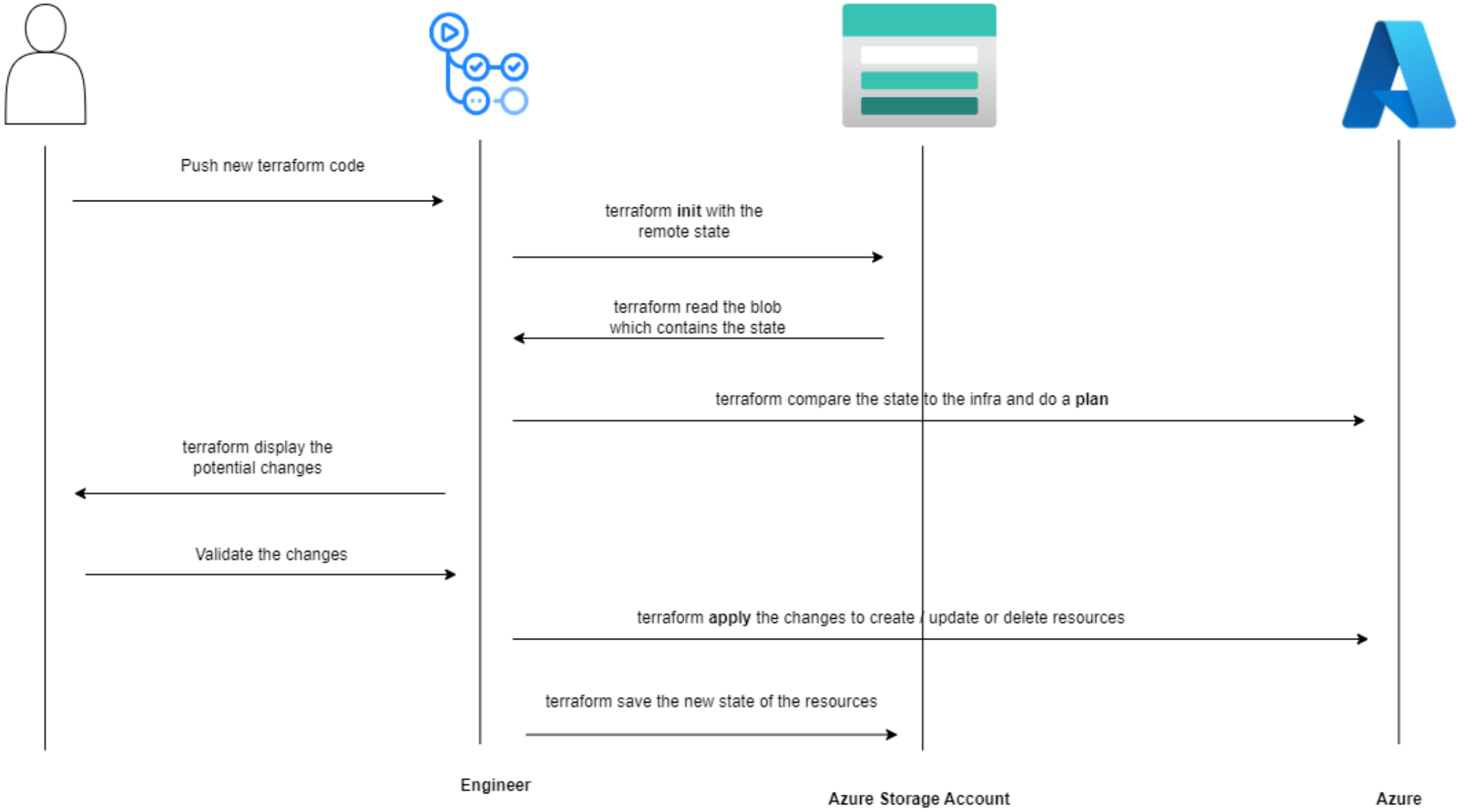
# Terraform State: Locally



# Terraform State: Remotely



# Terraform State: DevOps



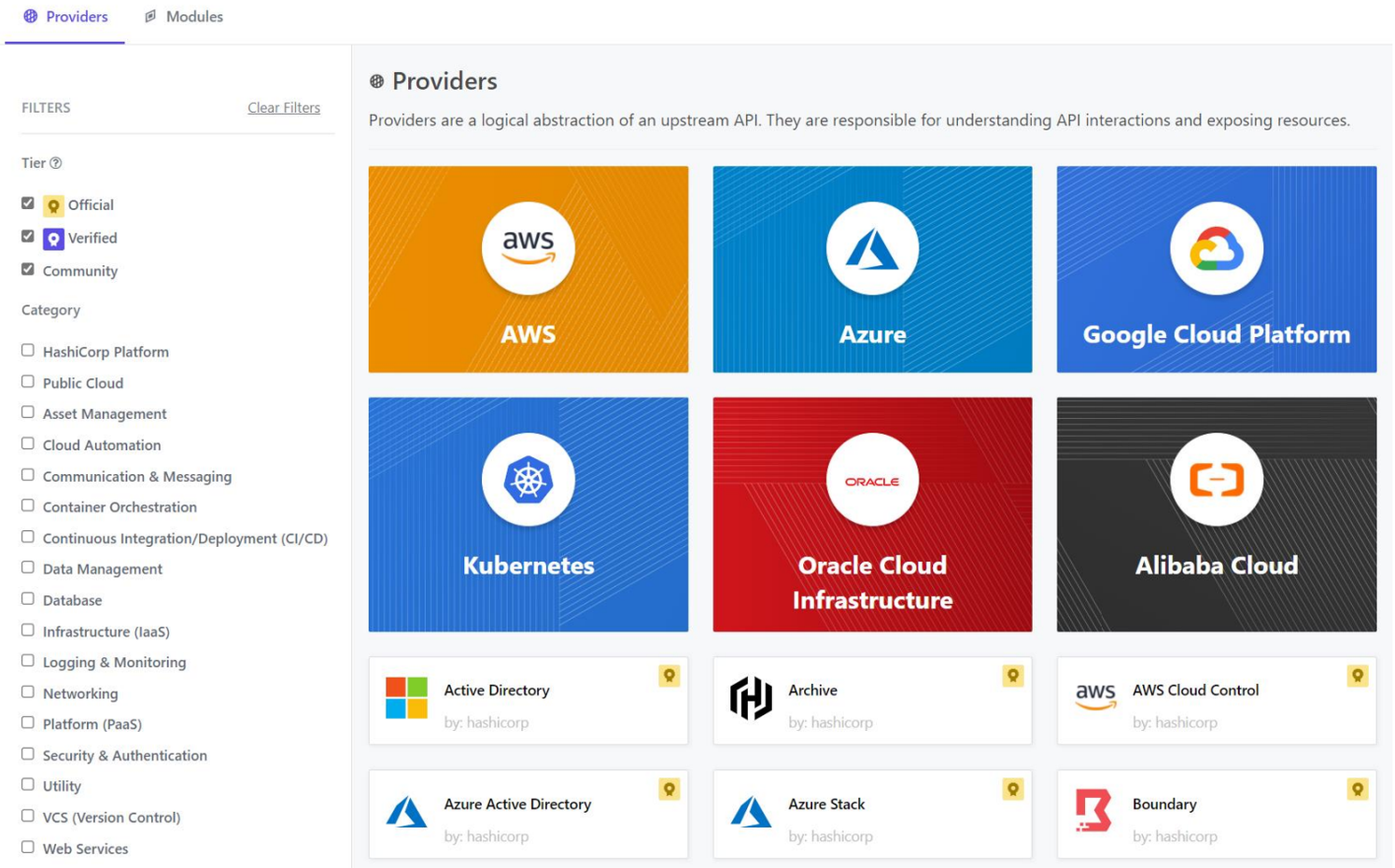
# Don’t touch the User Interface

- Keep the User Interface in **ReadOnly** mode
- If you modify a resource managed by Terraform you can end up with a **broken state**
- You will have to import each resource **manually** to the state, which can be really long and complex
- **terraform import** can be used to explicitly map actual resources to configurations.

```
terraform import -var-file ./env.dev.tfvars azurerm_resource_group.this
/subscriptions/733df85b-e409-4b3d-8ca2-abb90bb4a2c2/resourceGroups/my-
resource-group
```

# Providers

# Terraform Provider: Available ones




- https://registry.terraform.io/browse/providers

# Terraform Providers

- The terraform core program **requires at least one** provider to build anything.
- You can **manually configure** which **version**(s) of a provider you would like to use.
- If you leave this option out, Terraform will **default to the latest** available version of the provider.

```
provider "azurerm" {
  version = "=3.58.0"
}
```

# Configure azurerm provider

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "3.64.0"
    }
  }

  backend "local" { } #  backend "azurerm" {
}

provider "azurerm" {
  # Configuration options
}
```

# Authentication with AzureRm

# With Service Principal Name (SPN)

- Use a user account for **development purpose**
- Use a Service Principal Name (SPN) or Az CLI for **deployment purpose**
- The SPN should have **Contributor** role

```
az ad sp create-for-rbac --name <service_principal_name> --role Contributor --scopes /subscriptions/<subscription_id>
```

https://learn.microsoft.com/en-us/azure/developer/terraform/authenticate-to-azure?tabs=bash

# Service Principal Name directly in the code

- You can connect Terraform to Azure directly in the **code using the provider**
- SPN credentials in your code is a **bad practice**
- You will probably **forget to remove** them before pushing the code
- This is a **security** issue

```
provider "azurerm" {
  features {}

  subscription_id   = "<azure_subscription_id>"
  tenant_id         = "<azure_subscription_tenant_id>"
  client_id         = "<service_principal_appid>"
  client_secret     = "<service_principal_password>"
}
```

NOT RECOMMANDED

# Service Principal Name with environment variable

- You can connect Terraform to Azure using **environment variables**
- SPN credentials will be **only on your machine**
- Your code doesn't have **any credentials**
- Credential lifecycle is a problem with SPN, **you need to manage it**

```
export ARM_SUBSCRIPTION_ID="<azure_subscription_id>"
export ARM_TENANT_ID="<azure_subscription_tenant_id>"
export ARM_CLIENT_ID="<service_principal_appid>"
export ARM_CLIENT_SECRET="<service_principal_password>"
```

https://learn.microsoft.com/en-us/azure/developer/terraform/authenticate-to-azure?tabs=bash

# With Az CLI

- You can connect Terraform to Azure using **Az CLI directly**:

```
# Login to Azure
az login

# Show your account
az account show

# Select your subscription
az account set --subscription <your-subscription-id>
```

- Use it for **development purpose only**

# With Managed Identity

- Use managed identity declared in **Azure Active Directory**
- The token to authenticate to Azure is managed automatically by Azure Active Directory. You don’t need **to monitor the end of life** of your token

```sh
# sh
export ARM_USE_MSI=true
export ARM_TENANT_ID=00000000-0000-0000-0000-000000000000
export ARM_SUBSCRIPTION_ID=00000000-0000-0000-0000-000000000000
export ARM_CLIENT_ID=00000000-0000-0000-0000-000000000000

# PowerShell
$env:ARM_USE_MSI = true
$env:ARM_TENANT_ID = 00000000-0000-0000-0000-000000000000
$env:ARM_SUBSCRIPTION_ID = 00000000-0000-0000-0000-000000000000
$env:ARM_CLIENT_ID = 00000000-0000-0000-0000-000000000000
```

# With Open ID Connect (OIDC)

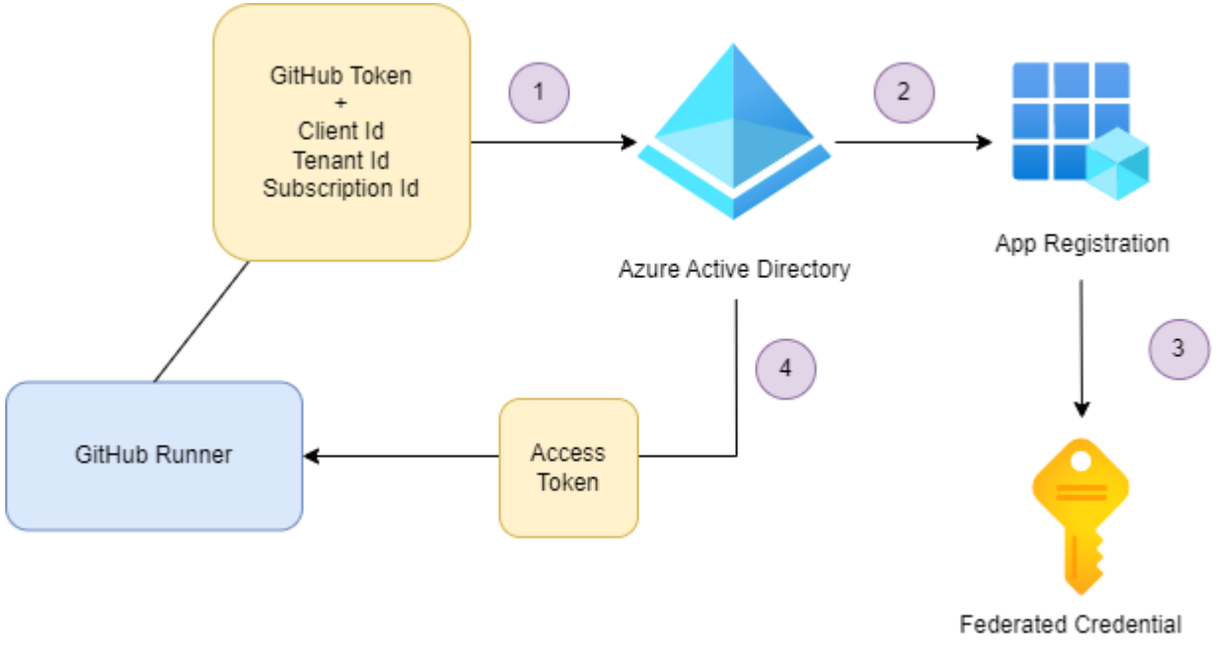
- Azure **manage** the credentials for you
- Generate credentials with **short lifetime**
- Avoid managing manually a **Service Principal Name** (SPN)
- Just need to:

  Create an **App Registration**

  Create **federated credentials** to give access to the repository

# OIDC: How it works?

- 1) GitHub Runner asks with a GitHub **token and an Azure AD identity** for an AAD token. Then token contains issuer, audience and subject.
- 2) Based on the token AAD search the associated identity in the **App Registration**
- 3) The App Registration has the **federated credential** for this repository
- 4) AAD return an access token, so the GitHub Runner can **modify Azure Resources**



# Hands On Lab!

# Connect to Azure Rm

- Create a new folder called **digital_infra_state**
- Create a provider.tf file and use use **Azure Rm** https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs
- Use **the latest version** of **Azure Rm**
- Use **local state**
- Run **terraform init** to validate that everything is **correctly initialized**

# Connect to Azure Rm

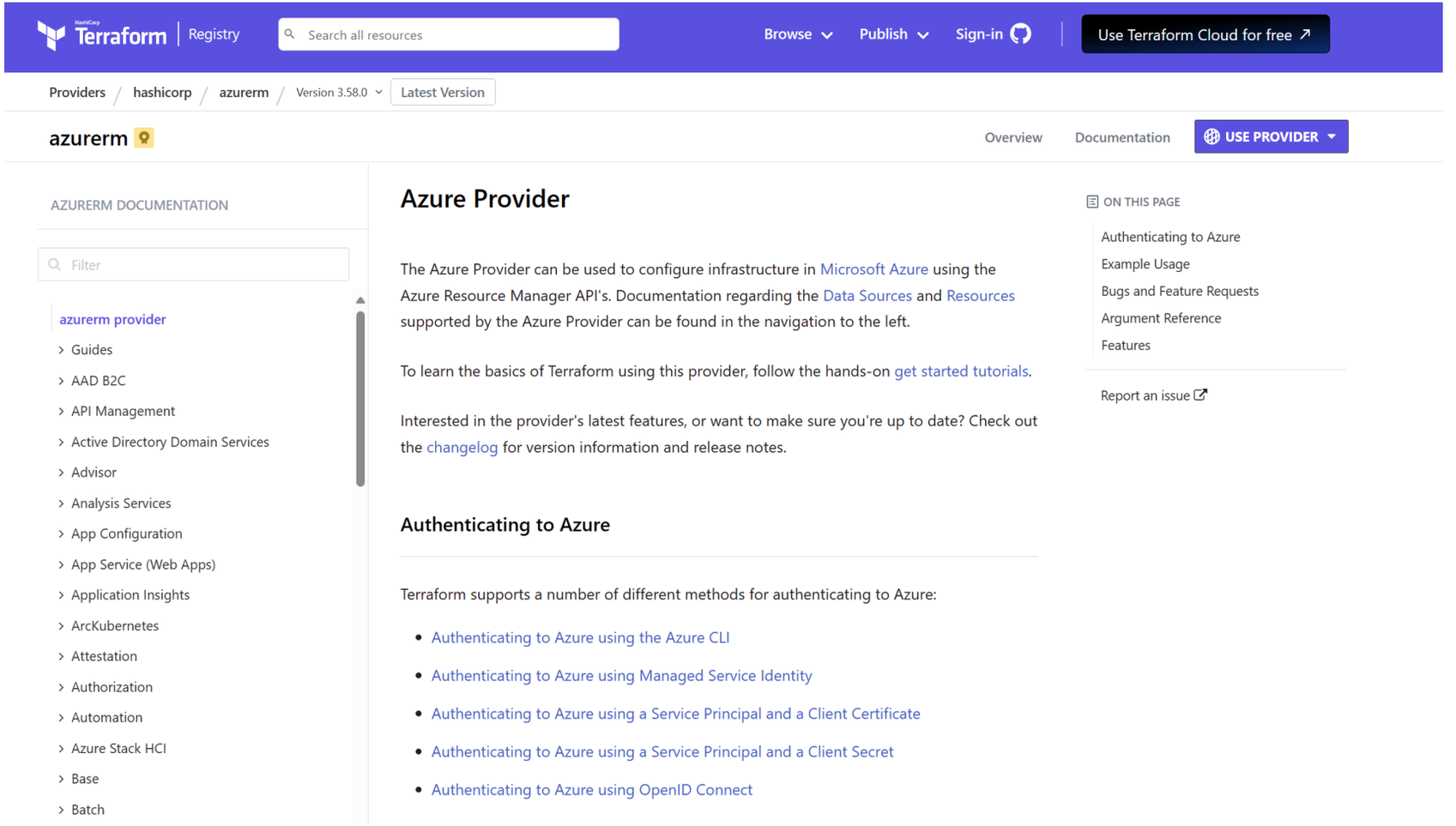
```
Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/azurerm from the dependency lock file
- Using previously-installed hashicorp/azurerm v3.59.0

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.
```

# Documentation usage: Overview

# Documentation

- Really good documentation
- The search bar can be a bit capricious
- Use your favorite search engine to find the resource. For instance, type: Resource group Terraform

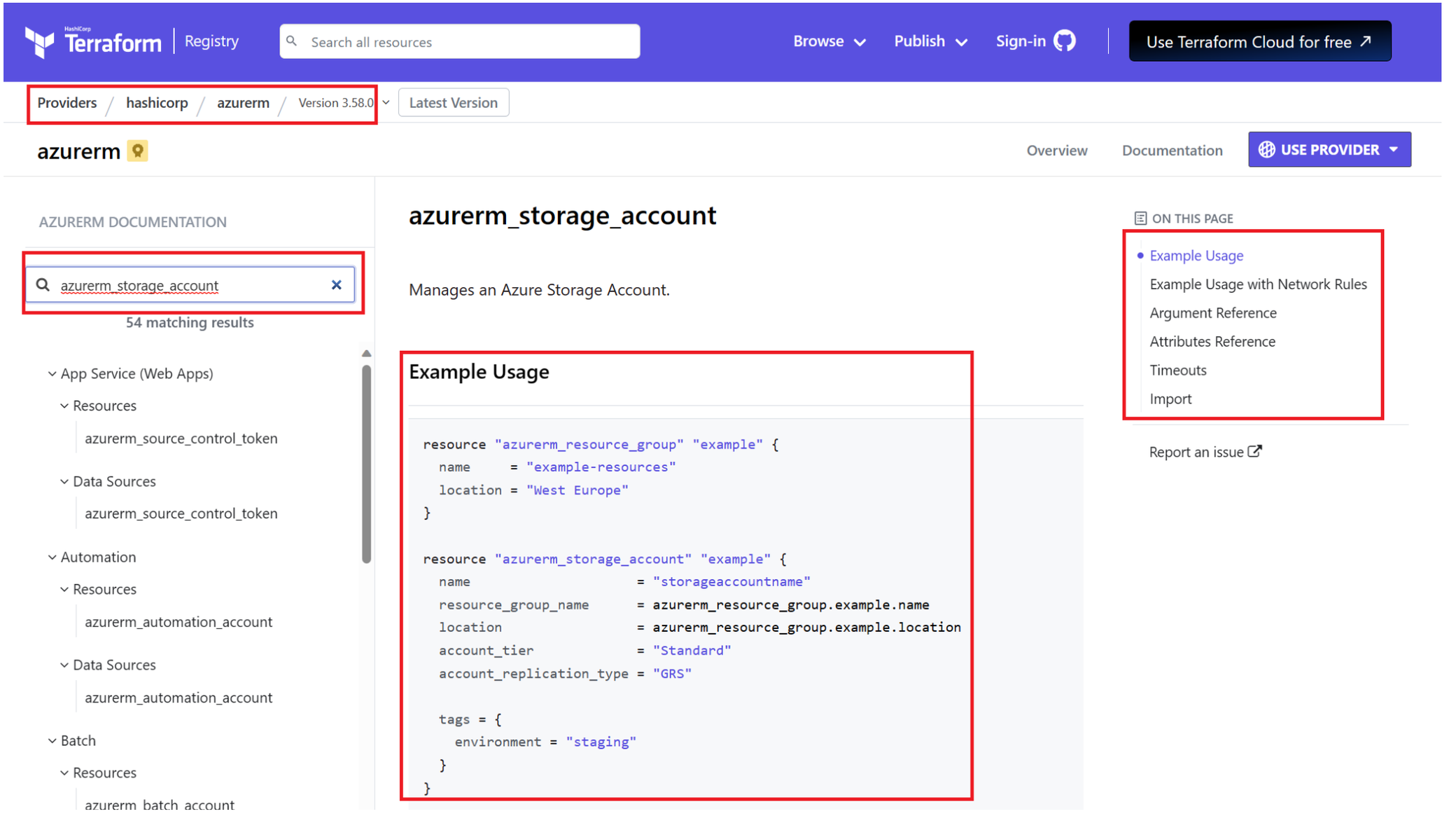
https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs



# Resource sections

- Always cut into several parts:
  - **Basic examples** to copy-paste
  - **Argument** References which are the available **properties** of the resource
  - **Attributes** References which are the available **outputs** of the resource
  - **Timeouts** properties that can be **optionally** specified
  - **Import** a resource to the **current state**

# Example



# Arguments

## Argument Reference

The following arguments are supported:

- `name` - (Required) Specifies the name of the storage account. Only lowercase Alphanumeric characters allowed. Changing this forces a new resource to be created. This must be unique across the entire Azure service, not just within the resource group.

- `resource_group_name` - (Required) The name of the resource group in which to create the storage account. Changing this forces a new resource to be created.

- `location` - (Required) Specifies the supported Azure location where the resource exists. Changing this forces a new resource to be created.

- `account_kind` - (Optional) Defines the Kind of account. Valid options are `BlobStorage`, `BlockBlobStorage`, `FileStorage`, `Storage` and `StorageV2`. Defaults to `StorageV2`.

> **NOTE:**
>
> Changing the `account_kind` value from `Storage` to `StorageV2` will not trigger a force new on the storage account, it will only upgrade the existing storage account from `Storage` to `StorageV2` keeping the existing storage account in place.

- `account_tier` - (Required) Defines the Tier to use for this storage account. Valid options are `Standard` and `Premium`. For `BlockBlobStorage` and `FileStorage` accounts only `Premium` is valid. Changing this forces a new resource to be created.

ON THIS PAGE



Report an issue

# Attributes

## Attributes Reference

In addition to the Arguments listed above - the following Attributes are exported:

- `id` - The ID of the Storage Account.
- `primary_location` - The primary location of the storage account.
- `secondary_location` - The secondary location of the storage account.
- `primary_blob_endpoint` - The endpoint URL for blob storage in the primary location.
- `primary_blob_host` - The hostname with port if applicable for blob storage in the primary location.
- `secondary_blob_endpoint` - The endpoint URL for blob storage in the secondary location.
- `secondary_blob_host` - The hostname with port if applicable for blob storage in the secondary location.
- `primary_queue_endpoint` - The endpoint URL for queue storage in the primary location.

ON THIS PAGE



Report an issue

# Hands On Lab!

# Create a resource group with terraform local state

- Back to your **digital_infra_state** folder

- Create a variable.tf file to define :
  - An environment which accept only “dev”, “stag”, “prod” with “dev” as default value
  - A domain name with default value to “state”
  - A location with a default value set to “westeurope”
  - A resource suffix with a default value

- Create a locals.tf file to define two **resource suffix** which are a concatenation of:
  - The environment
  - The first 2 letters of the location in lowercase
  - The first 3 letters of the domain in lowercase
  - The resource suffix

The resource suffixes should be in this format:

- **With** dashes (kebabcase): <env>-<domain>-<location>-<resource-suffix>
- **Without** dashes (lowercase): <env><domain><location><resource-suffix>

# Create a resource group with terraform local state

- Create a rg.tf file to define your resource group. Naming convension is: **rg-<your-resource-suffix-kebabcase>**

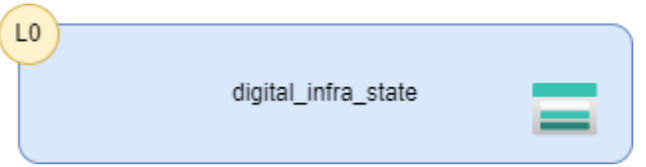
- Create an sto.tf file to define a basic **storage account:**
  The naming convention is **st<your-resource-suffix-lowercase>**
  Account tier: **Standard**
  Account replication type: **LRS**

- Inside the Storage Account create **one private container** called "**states**"

- Add tags "Domain" and "Environment" to all your resources.

- Use **local state**

- **Deploy** to Azure



# Backend definition

# Backend configuration

- The backend configuration for Azure Rm is a **Storage Account**
- The Storage Account contains a **container** with a **blob** which is the **tfstate**
- Which came first, the chicken or the egg?

# Connect to the Azure Rm backend in code

- Useful for testing purpose **BUT**:
- Avoid using this approach because your code must be agnostic to the targeted environment

NOT RECOMMANDED

```
terraform {
  backend "azurerm" {
    resource_group_name  = "digital_infra_state"
    storage_account_name = "stostate01"
    container_name       = "tfstates"
    key                  = "prod.terraform.tfstate"
  }
}
```

# Connect to the Azure Rm backend with backend config

- Use the **–backend-config** option to pass the Azure Rm settings
- This can be done for CI / CD
- **Use environment variables**

BEST PRACTICE

```
terraform init \
    -backend-config="resource_group_name=digital_infra_state" \
    -backend-config="storage_account_name=stostate01" \
    -backend-config="container_name=tfstates" \
    -backend-config="key=prod.terraform.tfstate"
```

# Several ways to use environment variables

# Pass variables to terraform commands

- Pass **-var** options on the command line

```
terraform apply -var="image_id=abc123"
```

# Pass variables to terraform commands

- Pass a file as options with **-var-file**

```
terraform apply -var-file="testing.tfvars"
```

- **.tfvars** list all variables to set for the deployment

```
diagnostic_settings_enabled = false
vnet_address_space_prefix = "10.0"
vnet_address_space_suffix = "0.0/16"
subnets = [
  {
    "application" = "prj1"
     "subnet_address_suffix" = "1.0/32"
  }
]
```

# Automatic load of variable files if present

Terraform also automatically loads a number of variable definitions files if they are present:

- Files named exactly **terraform.tfvars** or **terraform.tfvars.json**.
- Any files with names ending in **.auto.tfvars** or **.auto.tfvars.json**.

```
{
  "diagnostic_settings_enabled": false,
  "vnet_address_space_prefix": "10.0",
  "vnet_address_space_suffix": "0.0/16",
  "subnets": [
    {
      "application": "prj1",
      "subnet_address_suffix": "1.0/32"
    }
  ]
}
```

# Other ways to pass variables

- Environment variables (“TF_VAR_[variable]”, e.g. “TF_VAR_rg_id” for “rg_id”)
- Default Config - default value in **variables.tf**
- User manual entry - if not specified, prompt the user for entry

```
export TF_VAR_image_id=ami-abc123
terraform plan
```

# Order of precedence of variables

- Once you have some variables defined, you can **set and override** them in **different ways.**

- Level of precedence for each method (1=highest, 7=lowest):
  1. **-var** and **-var-file** options on the command line
  2. Any ***.auto.tfvars** or ***.auto.tfvars.json** files, processed in lexical order of their filenames.
  3. The **terraform.tfvars.json** file, if present.
  4. The **terraform.tfvars** file, if present.
  5. Environment variables ("**TF_VAR_[variable]"**, e.g. **"TF_VAR_rg_id"** for **"rg_id"**)
  6. Default Config - default value in **variables.tf**
  7. User **manual entry** - if not specified, prompt the user for entry

# Hands On Lab!

# Create a Vnet and Subnet

- Create a new folder called **digital_infra_network**

- Based on the same model of the previous lab:
  Create a **variable.tf**
  Create a **locals.tf**
  Create a **provider.tf**
  Create an **rg.tf** for a new resource group
  The domain will be the **“network”**, so find a diminutive to it

- Use **remote state** and **store it** inside the Storage Account containers “states” created in the previous lab in this path: **network/terraform.state**

# Create a Vnet and Subnet

- Create a **vnet.tf** file to define a VNet with an address range of: 10.0.0.0/16

- Create a **subnets.tf** file to define 2 subnets:
  - PaasSubnet: 10.0.0.0/24
  - AppServiceSubnet: 10.0.1.0/28 with a **delegation** to "`Microsoft.Web/serverFarms`" and and **action** to "`Microsoft.Network/virtualNetworks/subnets/action`"

- Expose the VNet and Subnet Ids in an **outputs.tf** file

- Create a folder called **env-vars** for your environment variables and one folder for the **dev** environment. Create an **env.tfvars** file inside the **dev** folder to specify the values for the IP ranges for the VNet and the Subnets.

- Deploy only the **dev** environment

# Meta-Arguments

# Meta-Arguments: depends_on

- **Explicitly** specifying a dependency
- Necessary when a resource or module relies on some other resource's behavior but doesn't access to any of that resource's data in its arguments.

```
module "apim" {
 source   = "./modules/api_management"
 rg_name  = azurerm_resource_group.apifunc_rg.name
 location = azurerm_resource_group.apifunc_rg.location
}
```

```
module "cdh_api_endpoints" {
 source  = "./modules/cdh_api_endpoints"
 rg_name = azurerm_resource_group.apif_rg.name
 apim_name                   = "bmw-cdh-core-api"
 api_management_name         = var.api_apim_name
 api_management_backend_name = "cdh-api"
 depends_on = [
   module.apim
 ]
}
```

# Meta-Arguments: lifecycle

- Control lifecycle of a resource using arguments within lifecycle block:
  **create_before_destroy:** bool, default false, if destroy is required, it is done after creating the new resource

  **prevent_destroy:** bool, default false, plans resulting in a destroy will fail

  **ignore_changes:** List of feature names (string), to be ignored, if different
    Common scenarios to ignore later changes of initial values
      **default secrets in key vault**
      **login of a sql server**
      **app settings of a web app**

- Only allows literal elements, no variables / expressions

```
resource "azurerm_function_app" "this" {
  name = var.function_app_name
  # ...

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

# Meta-Arguments: provider

- Select non-default **provider** to be used
- Example: **multiple AzureRm provider** for different subscriptions

```hcl
# Provider for current subscription
provider "azurerm" {
  features {}
}


# Provider for non-default Subscription
provider "azurerm" {
  alias                      = "resource_subscription"
  subscription_id            = var.resource_subscription_id
  skip_provider_registration = true
  features {}
}
```

```hcl
# RG in Default-Subscription
resource "azurerm_resource_group" "this" {
  name     = var.api_keyvault_resourcegroup
  location = var.location
}

# RG in non-default subscription
resource "azurerm_resource_group" "this" {
  provider = azurerm.resource_subscription
  name     = var.api_keyvault_resourcegroup
  location = var.location
}
```

# Meta-Arguments: count

- Enables creation of **0..n similar objects**

- Count cannot be used simultaneously with meta-argument for_each.

Usage example:
- Zero-Based index number: count.index
- Individual instance: <TYPE>.<NAME>[<INDEX>]  or module.<NAME>[<INDEX>]
- In the example below **Count = 0** means **skip creation**

```
resource "azurerm_monitor_metric_alert" "this" {
  count               = var.add_default_alerts ? 1 : 0
  name                = "Failed-Requests-Alert-${var.function_app_name}"
  resource_group_name = var.rg_name
  # ...
}
```

# Meta-Arguments: for_each

- Enables creation of **0..n similar objects**
- for_each cannot be used simultaneously with meta-argument count
- Use for_each instead of count for loop, this will avoid to have terraform state lost

Usage example:

- Current object: each.key (map key / set member), each.value (map value)
- Individual instance: <TYPE>.<NAME>[<KEY>]  or module.<NAME>[<KEY>]

```
resource "azurerm_resource_group" "this" {
  for_each = {
    a_group       = "eastus"
    another_group = "westus2"
  }
  name     = each.key
  location = each.value
}
```

# Use for_each for the loops

- Use **for_each** instead of **count** for looping over a list of element.
- This will allow terraform to **keep tracking** existing resources and not recreating them

# Hands On Lab!

# Create an App Service

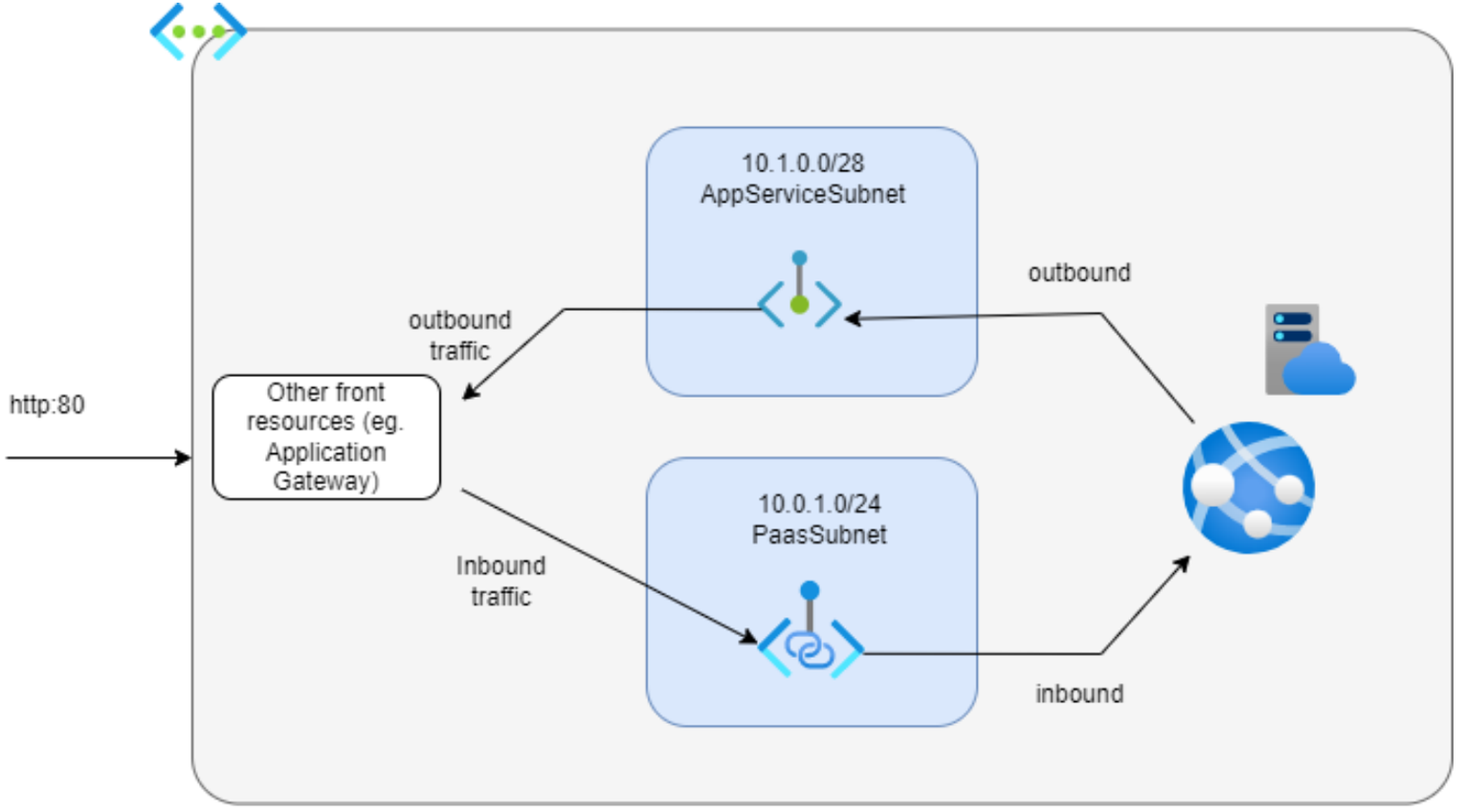
- Create a new folder called **prj1_infra_app**

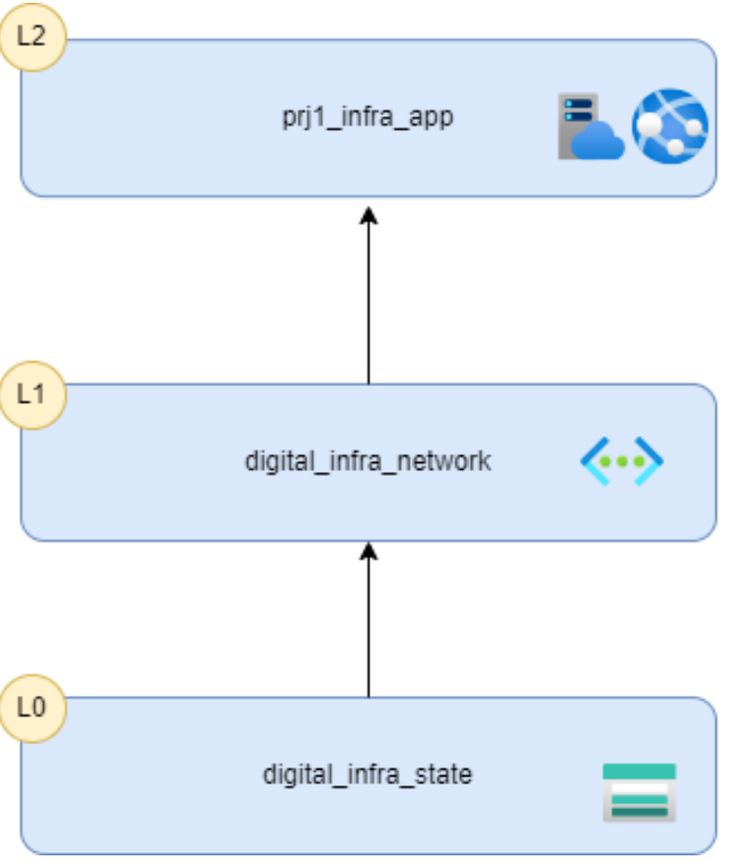
- Based on the same model of the previous lab:
  Create a **variable.tf**
  Create a **locals.tf**
  Create a **provider.tf**
  Create an **rg.tf** for a new resource group
  The domain will be the **“prj1”** and add a new variable for the application name and call it whatever you want but limit the characters to 3.

- Use **remote state** and **store it** inside the Storage Account containers “states” created in the previous lab in this path: **prj1/<application name>/terraform.state**

# Create an App Service

- The goal of this lab is to consume the state network state and activate the VNet integration and private endpoint for the App Service.



# Create an App Service

- Create an **app.tf** and define:
  - An App Service Plan with Basic SKU
  - An App Service for Linux

- Create a **data.tf** file and use the `terraform_remote_state` resource to connect to the network state with the correct credential methodology:
- https://developer.hashicorp.com/terraform/language/settings/backends/azurerm

- To be able to consume the state from another project you have to give the **Storage Blob Data Owner** role on the storage account to manage the states for Azure AD authenticated users:
  https://developer.hashicorp.com/terraform/language/settings/backends/azurerm

  Logout and login again to Azure CLI to refresh the token.

# Create an App Service

- In your **locals.tf** get the subnet ids for the app service and paas subnets.

- In the **app.tf** use the `azurerm_app_service_virtual_network_swift_connection` resource to connect the app service to the app service subnet:

https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/app_service_virtual_network_swift_connection

# Create an App Service

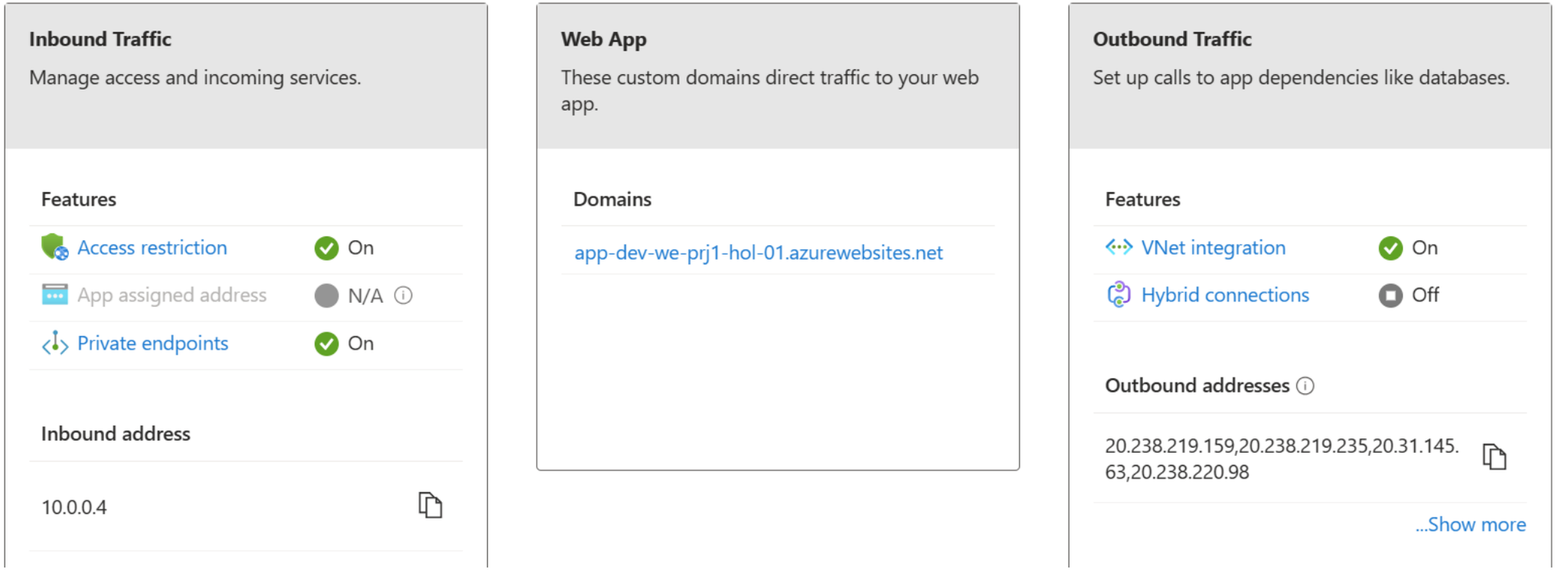
- Create a **pe.tf** file to define the private endpoint
- https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/private_endpoint

- The subnet Id should be the Paas Subnet id

- For the private service connection, you just have to define:
  - A name
  - The private connection resource id, which is the id of your App Service
  - Sub resources name array should contains only “sites”
  - Set the manual connection boolean to false.

# Create an App Service

If you succeed you should in the Networking section of your App Service these features On:



# Terraform Modules

# What is a module?

- A module is a **reusable** Terraform template
- Allow re-use of **certified** and **standardized** infrastructure as **code**
- **Can be published** at Public Module Registry at https://registry.terraform.io/
- **Private Repo** can be established using GitHub, Bitbucket , S3 buckets,….

# Structure of a module

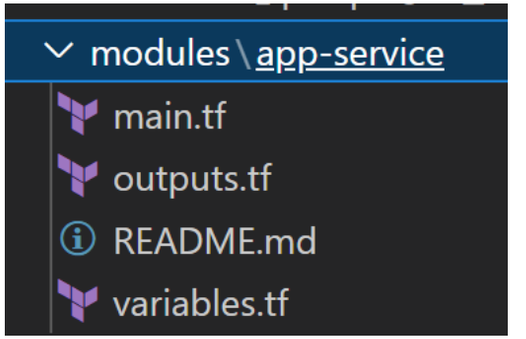

- **File**: variables.tf
- Parameters for module

- **File**: main.tf
- Resources and Data blocks

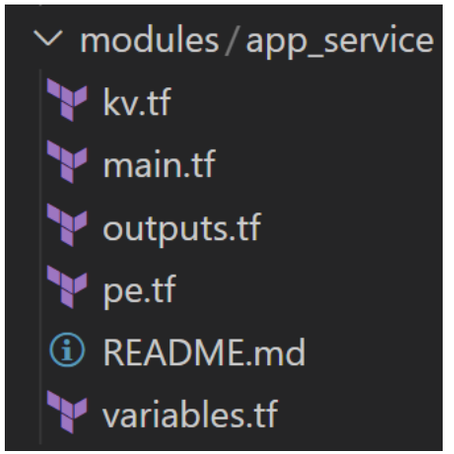
- **File**: outputs.tf
- Return results, others may reuse for their processing

# Structure of a module

- If you follow the **convention**, you should have this kind of structure:




- For more clarity you can **split the main.tf** file to avoid having one big file with everything defined.
- **Avoid** using **data sources** inside a module this can create dependency resolution **issues** for terraform



# Module supported sources

| What | How |
| --- | --- |
| Local Paths | module "alert_group" {<br>  source = "./modules/alert_action_group“<br>} |
| Terraform Registry | module "consul" {<br>  source = "hashicorp/consul/aws"<br>  version = "0.1.0"<br>}<br><br>module "consul" {<br>  source = "app.terraform.io/example-corp/k8s-cluster/azurerm"<br>  version = "1.1.0"<br>} |
| GitHub | module "consul" {<br>  source = "app.terraform.io/example-corp/k8s-cluster/azurerm"<br>  version = "1.1.0"<br>}<br><br>module "consul" {<br>  source = "git@github.com:hashicorp/example.git"<br>} |
| Generic git repository | module "vpc" {<br>  source = "git::https://example.com/vpc.git"<br>}<br><br>module "storage" {<br>  source = "git::ssh://username@example.com/storage.git"<br>} |

# Pass parameters to a module

- Run a **terraform init again to install** the local module
- Example of using a module locally:

```
module "resource_name" {
  source = "./modules/resource_folder_name"
  # Pass the parameters based on the variables.tf file of your module
  sku = "Basic"
  suffix_name = "dev-app-01"
}
```

# Refactoring into a module

- Modules are **referenced in Terraform state**
- Refactoring into a modules has an **impact** on the **state**

```
Plan: 4 to add, 0 to change, 4 to destroy.
```

- Resource names can be **changed** or **refactored** into modules
- **Define a tf file** (moved.tf for instance) when extracting a module during refactoring

```
moved {
  from = azurerm_storage_account.myStorageAccount
  to   = module.storageAccount.azurerm_storage_account.storeacc
}
```

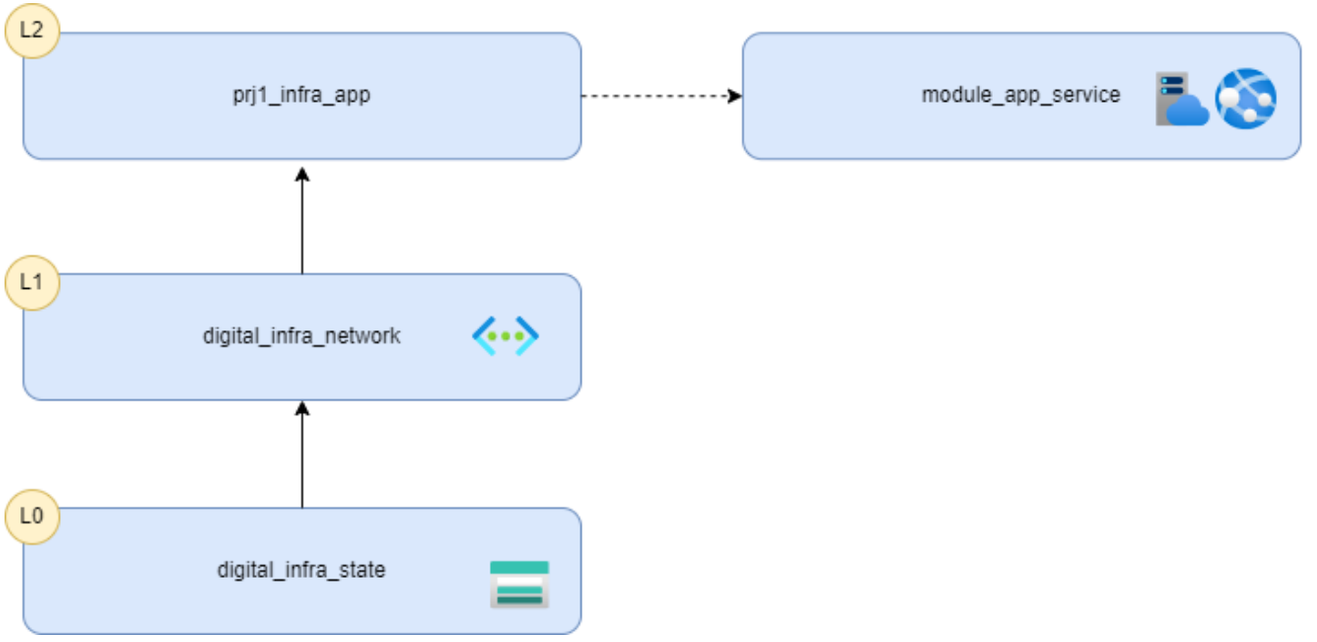
# Refactoring into a module

- Map the **old resource name** to the **new one** so terraform can **update the state**
- **Use terraform plan** to detect the old new naming
- If your moved declarations are correct your **plan will not** have any resources creation or deletion
- **After** this process you can remove your .tf file with all the **moved** declarations

# Hands On Lab!

# Migrate the App Service to a module

- Back in your **prj1_infra_app** create a new folder called **modules** and inside another one called **app_service**
- The goal is to provide a module for app service that manage the VNet integration.
- In this module, create a **main.tf** and put the **app.tf** file content into it.
- Create a **variables.tf** and an **outputs.tf**
- Move also the **pe.tf** inside this module



# Migrate the App Service to a module

- Update all the local properties in the modules
- Make the app service plan sku a parameter of your module
- Add two outputs to your module: the app service id and app service default hostname
- **Reference** the new module in the **app.tf** of your **prj1_infra_app** folder
- Make sure to migrate the module **without** creating or deleting any resource
- You may have to use a **lifecycle** block
- Run **terraform plan** and **terraform apply** you should have no changes

# Provisionners

# Provisionners

- Provisioners provide access to **non-terraform commands** and aren't **fully managed by it's declarative syntax or state.**

Examples:

- **local-exec** – run a command locally
- **remote-exec** – run a ssh command on a remote system
- **file** – copy files or directories to remote machine

- All provisioners support at least the following arguments:
  - **when** = destroy – execute only on destroy
  - **on_failure** = fail / continue – default is fail

# local-exec

- Execute scripts or other programs on the host executing terraform.

| Argument | Type | Description |
|---|---|---|
| command | string, required | • Command to execute<br>• Evaluated in a shell, and can use environment variables or Terraform variables |
| working_dir | string, optional | • Working directory where the command will be executed |
| interpreter | string list, optional | • List of interpreter arguments used to execute the command.<br>• First argument is the interpreter itself, as relative path to the current working directory or as an absolute path.<br>• The remaining arguments are appended prior to the command.<br>• Example: ["/bin/bash", "-c", "echo foo"]<br>• If interpreter is unspecified, sensible defaults will be chosen based on the system OS |
| environment | object, optional | • Block of key value pairs representing the environment of the executed command. |

# local-exec

- Example: command + interpreter

```
resource "null_resource" "this" {
  provisioner "local-exec" {
    command = "Get-Date > completed.txt"
    interpreter = ["PowerShell", "-Command"]
  }
}
```

- Example: command + environment

```
resource "null_resource" "this" {
 provisioner "local-exec" {
    command = "echo $FOO $BAR $BAZ >> env_vars.txt"

    environment = {
      FOO = "bar"
      BAR = 1
      BAZ = "true"
    }
  }
}
```

# remote-exec

- Execute scripts or other programs on a remote system.
- Only one of the following arguments allowed:

| Argument | Type | Description |
|---|---|---|
| inline | string list | • List of executed commands in specified order |
| script | string | • Path (relative or absolute) of a local script<br>• Local script will be copied to the remote resource and then executed |
| scripts | string list | • List of paths (relative or absolute) of a local script<br>• Local scripts will be copied to the remote resource and then executed in specified order |

# remote-exec

- Execute scripts or other programs on the remote system

- Examples:
  - Change some permissions and ownership
  - Run a script with some environment variables

```
resource "some_resource" "this" {
  ...
    provisioner "remote-exec" {
      inline = [
        "sudo chown -R ${var.admin_username}:${var.admin_username}
/var/www/html",
        "chmod +x *.sh",
        "PLACEHOLDER=${var.placeholder} WIDTH=${var.width}
HEIGHT=${var.height} PREFIX=${var.prefix} ./deploy_app.sh",
      ]
    }
...
}
```

# file

- Copy files or directories to a remote system

| Argument | Type | Description |
|---|---|---|
| source | String | • Source file or directory |
| content | String | • Direct content to copy on the destination<br>• If destination is a file, the content will be written on that file. |
| destination | string | • Destination path to write to on the remote system<br>• Will be evaluated by the remote system, rather than by Terraform itself |

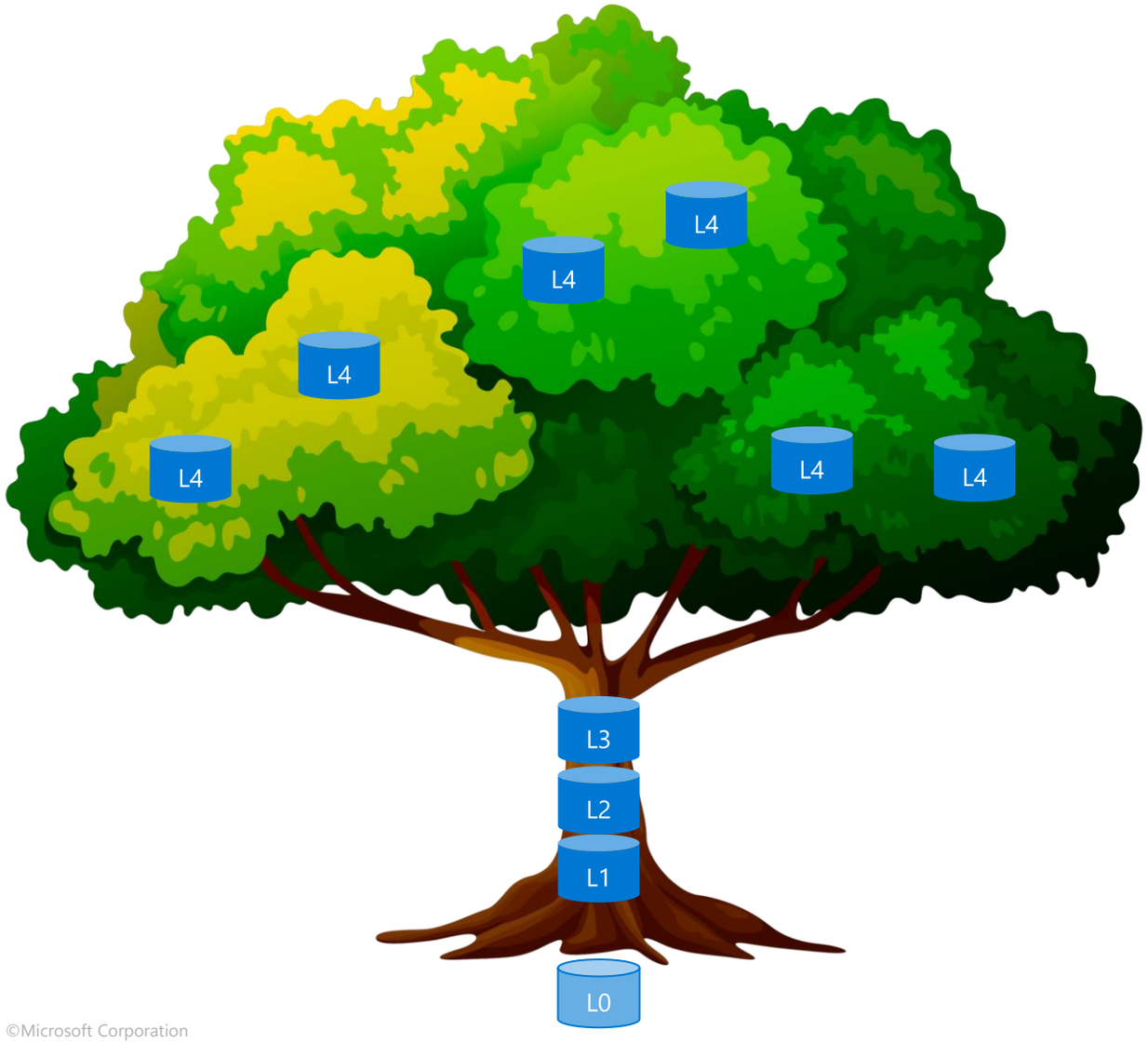
# file

- **Copies files or directories** onto the remote system
- Support for **SSH** and **WinRM connections**

```
provisioner "file" {
  source = "files/"
  destination = "/home/${var.admin_username}/"
  connection {
    type     = "ssh"
    user     = "${var.admin_username}"
    password = "${var.admin_password}"
    host     = "${azurerm_public_ip.catapp-pip.fqdn}"
  }
}
```

# Architecture

# Layered Deployment



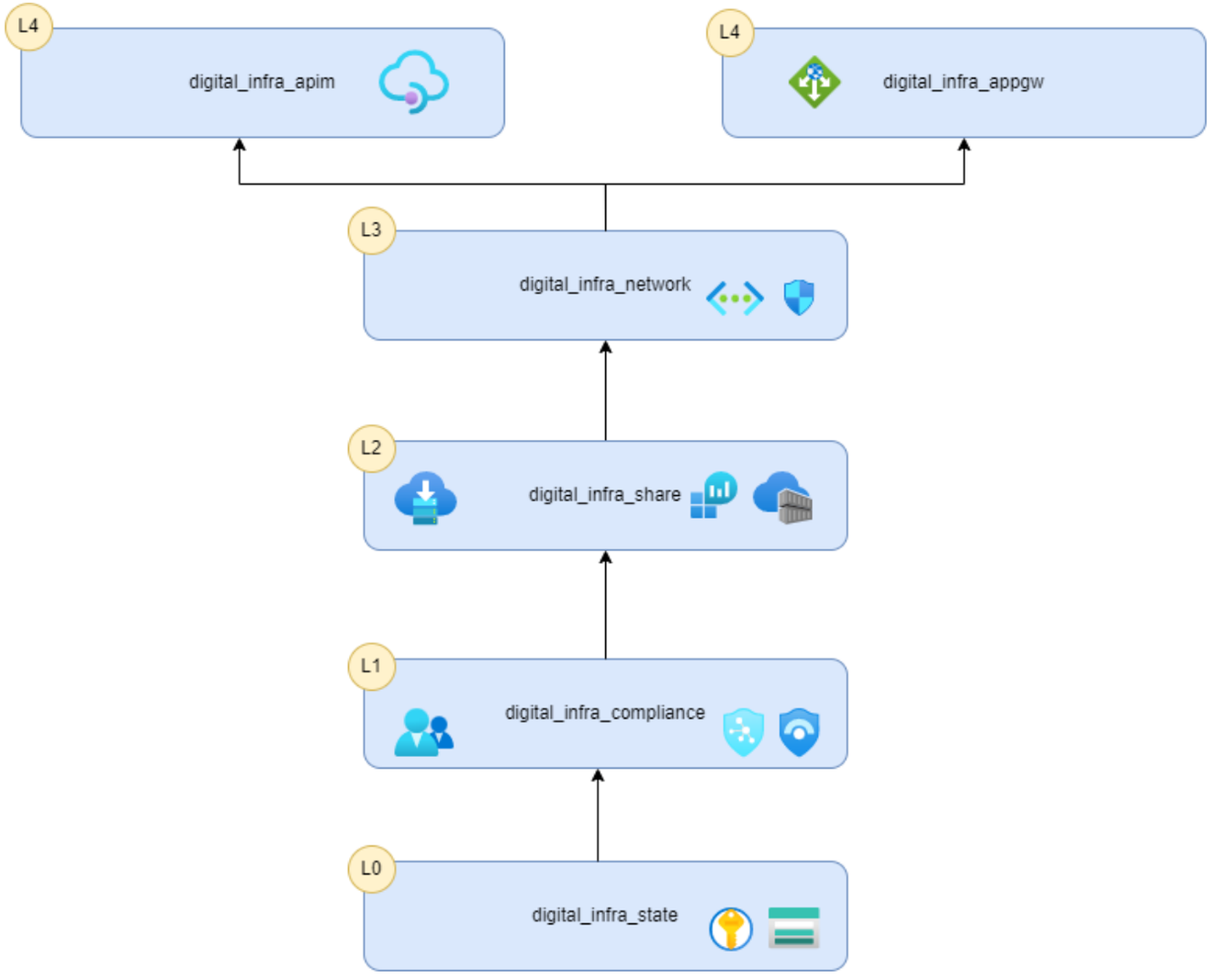

| Layer | Components |
|---|---|
| 0 | Bootstrap with init script / DevOps Agent |
| 1 | Security & Compliance |
| 2 | Core Services & Network |
| 3 | Shared Services (Compute, Data Server) |
| 4 | Applications (DB, WebApp, Queue,…) |

# Layered Deployment

- Reduce the scripts to a **specific scope**
- **Speed** things up
- **Split responsibilities** and permissions between teams.
- Each level has **its own state**

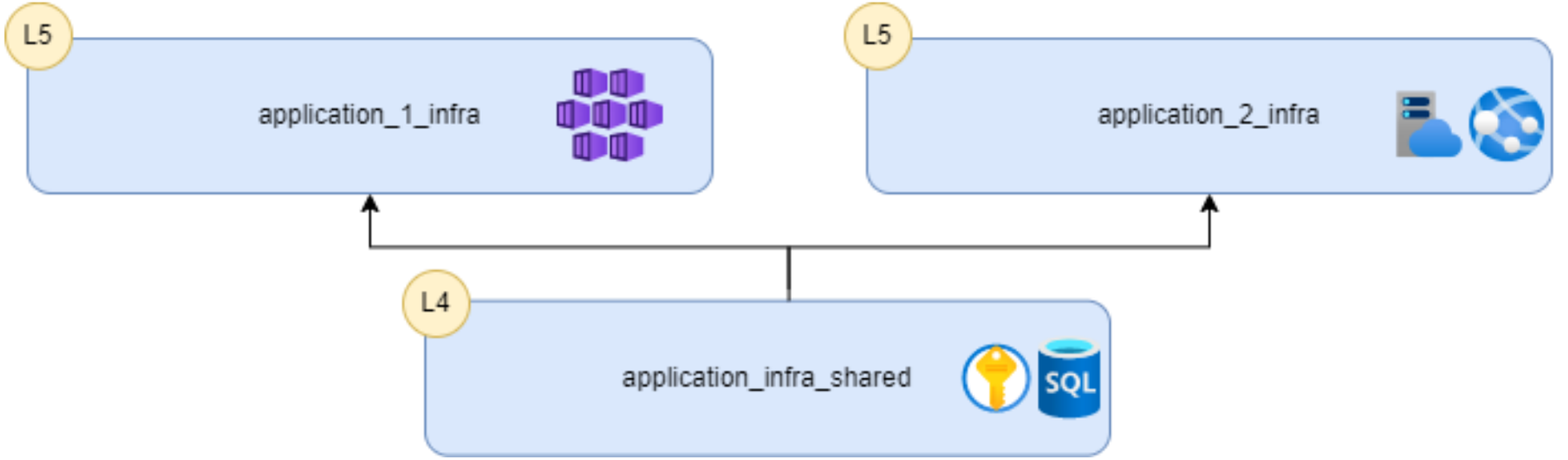
# Layered Deployment: Example

- Every layer has his own tfstate **except the first one**
- Each layer has **outputs exposed** to the next layer.
- You can add **more layers** to fit your needs
- Example:
  - L1 expose outputs to L2 / L3…
  - L3 can consume outputs from L2 / L3

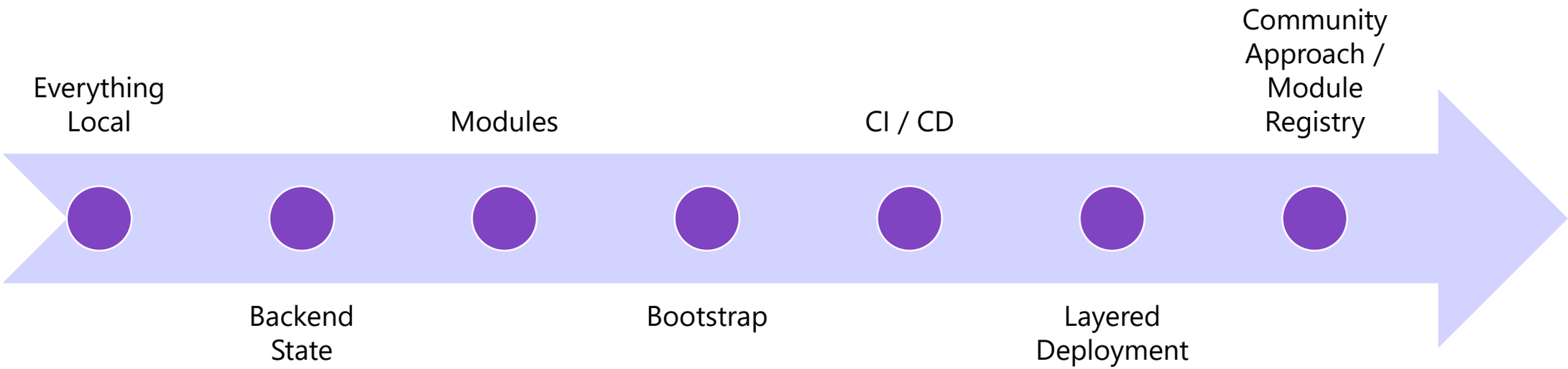


# Layered Deployment: Example

- **Applications layers** can be splitted to share resources
- Only when **business requirements** are compatible



# Layered Deployments and maturity



# DevOps

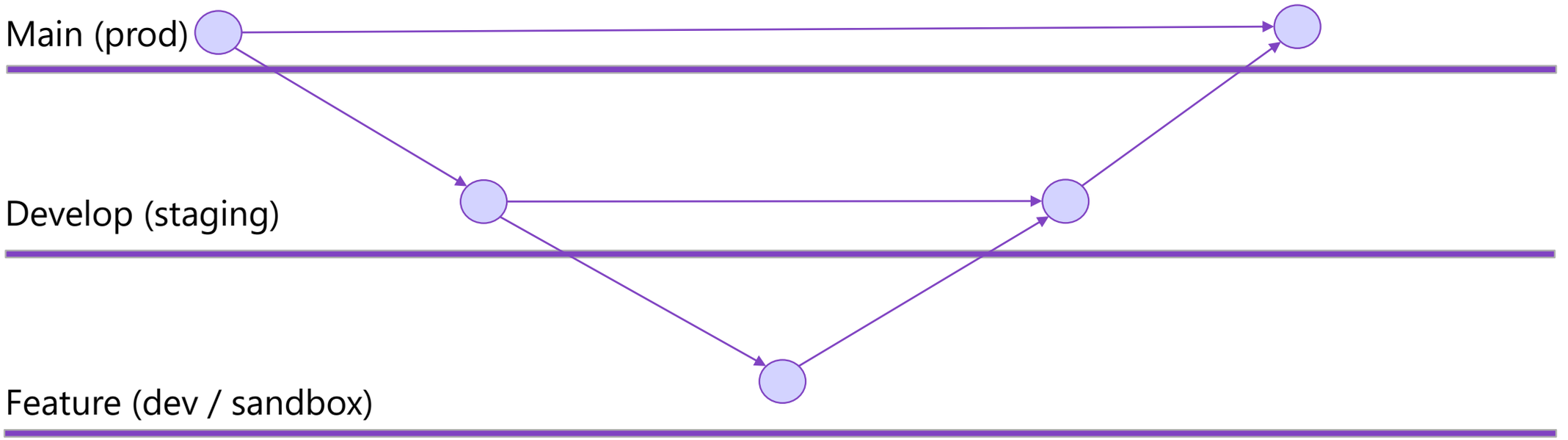
# Branch & deployment strategy

- Look at 10 projects you will find 10 different deployments
- The **best way** depends on **your project**
- Do a **brainstorming** before starting your project

# Decide on a branch strategy

- **Main** branch is rolling out the code to **production** infrastructure
- **Develop** branch is rolling out the code to **staging** infrastructure
- **Feature** branch can be manually rolling out in a sandbox / dev subscription
- When the project grows, feature branch are rolling out using **CI/CD** in a dev environment **like the other branches.**

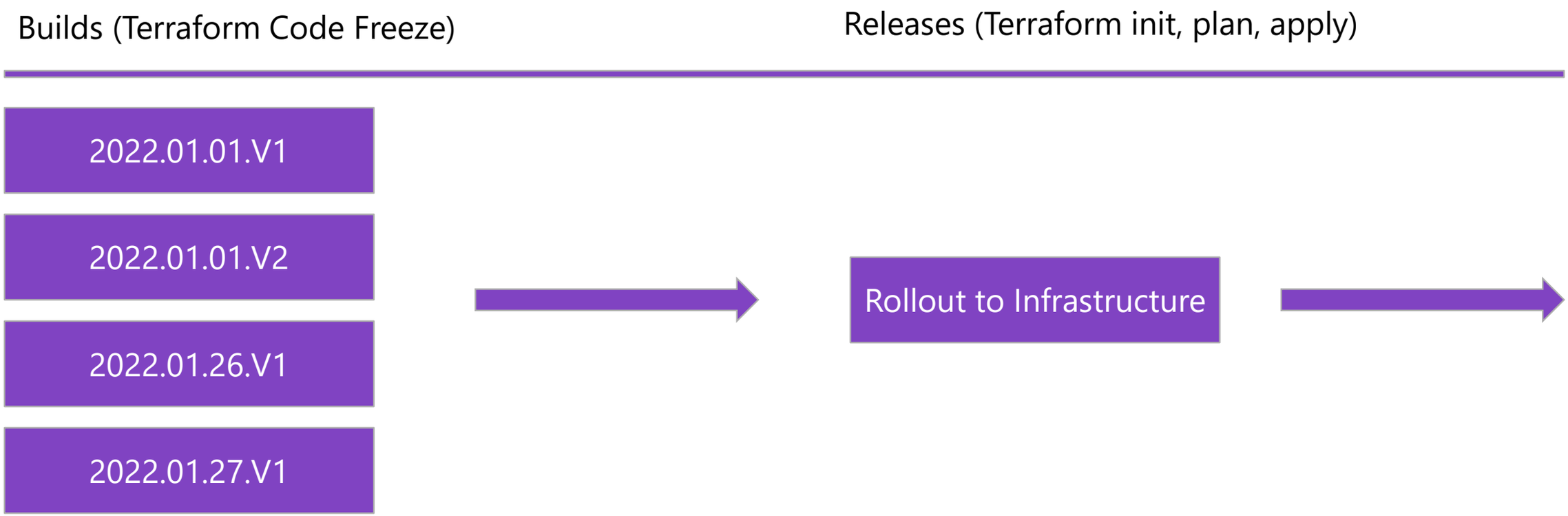
# Branching Strategy



# Decide on a deployment strategy

- **Create stages** with own environment variables
- Use **managed identity** for each environment
- **Do not define** backend configuration in your terraform code
- Create **flexible** pipelines or workflows, so a new stage for a new environment can be added **easily**
- Test your Terraform Code **before rollout** in a dev environment
- When working on **new features** be careful, when you deploy in a testing environment to not erase the deployment made by your teammate
- When the project grows, because of the **dependencies**, it becomes more and more **complicated** to work outside of a deployment environment common to the whole team

# Deployment strategy



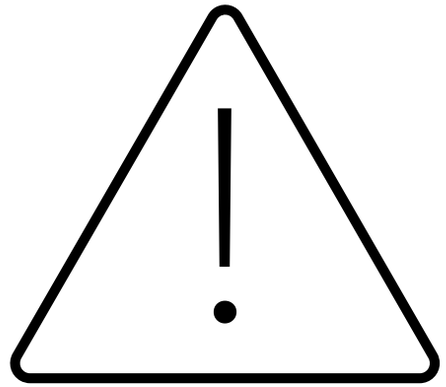
# Build strategy

- **Freeze** your Terraform (package, tag, versioning)
- **Dynamic** Terraform **plan** will be done during release
- Let you **downgrade** environments as needed
- Output your plan in a file to have something **static to deploy**

# Release strategy

# Terraform Rollout

- To change some resources **Terraform deletes** them and **recreates** them
- **Always check the plan** and test it on non-productive data



# Many tools to start

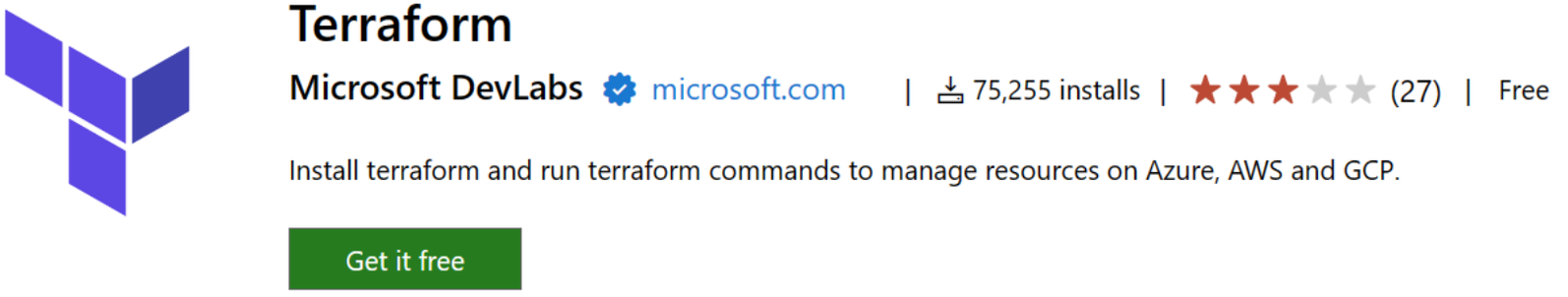
- Azure DevOps,  GitHub Actions, etc..



- Of course, you can reproduce the commands run by the tasks and actions directly using **basic command lines**

# Azure DevOps

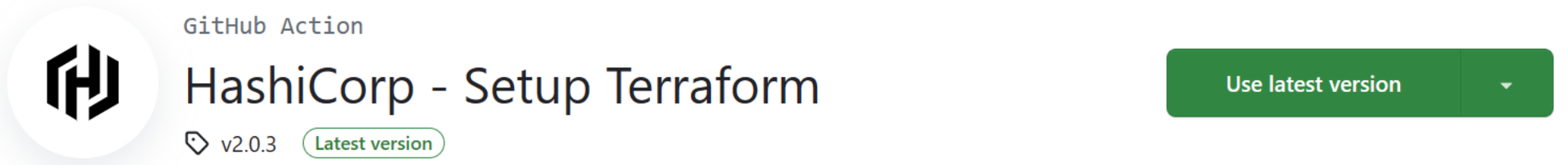
- Pre-build **tasks** are available to help you, install terraform and run commands in your pipelines




https://marketplace.visualstudio.com/items?itemName=ms-devlabs.custom-terraform-tasks

# GitHub Actions

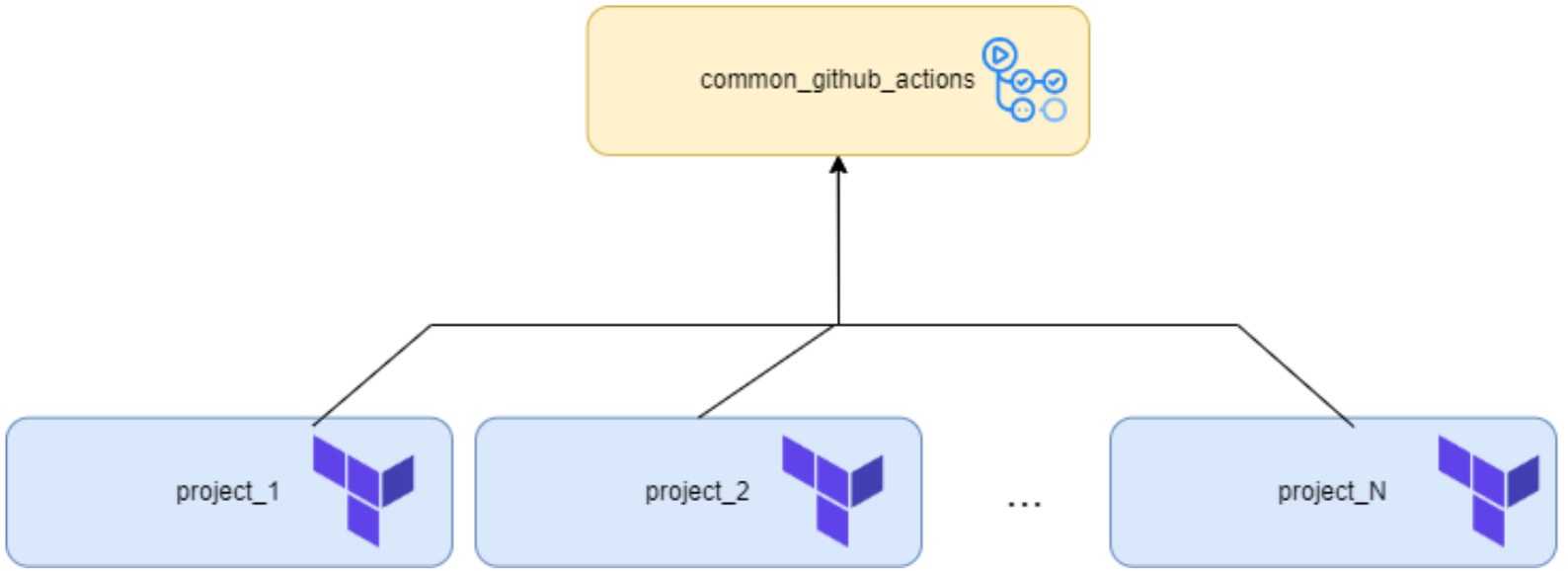
- Pre-build **actions** are available to help you, install terraform and run commands in your workflows.




https://github.com/marketplace/actions/hashicorp-setup-terraform

# Mutualize your GitHub Workflows

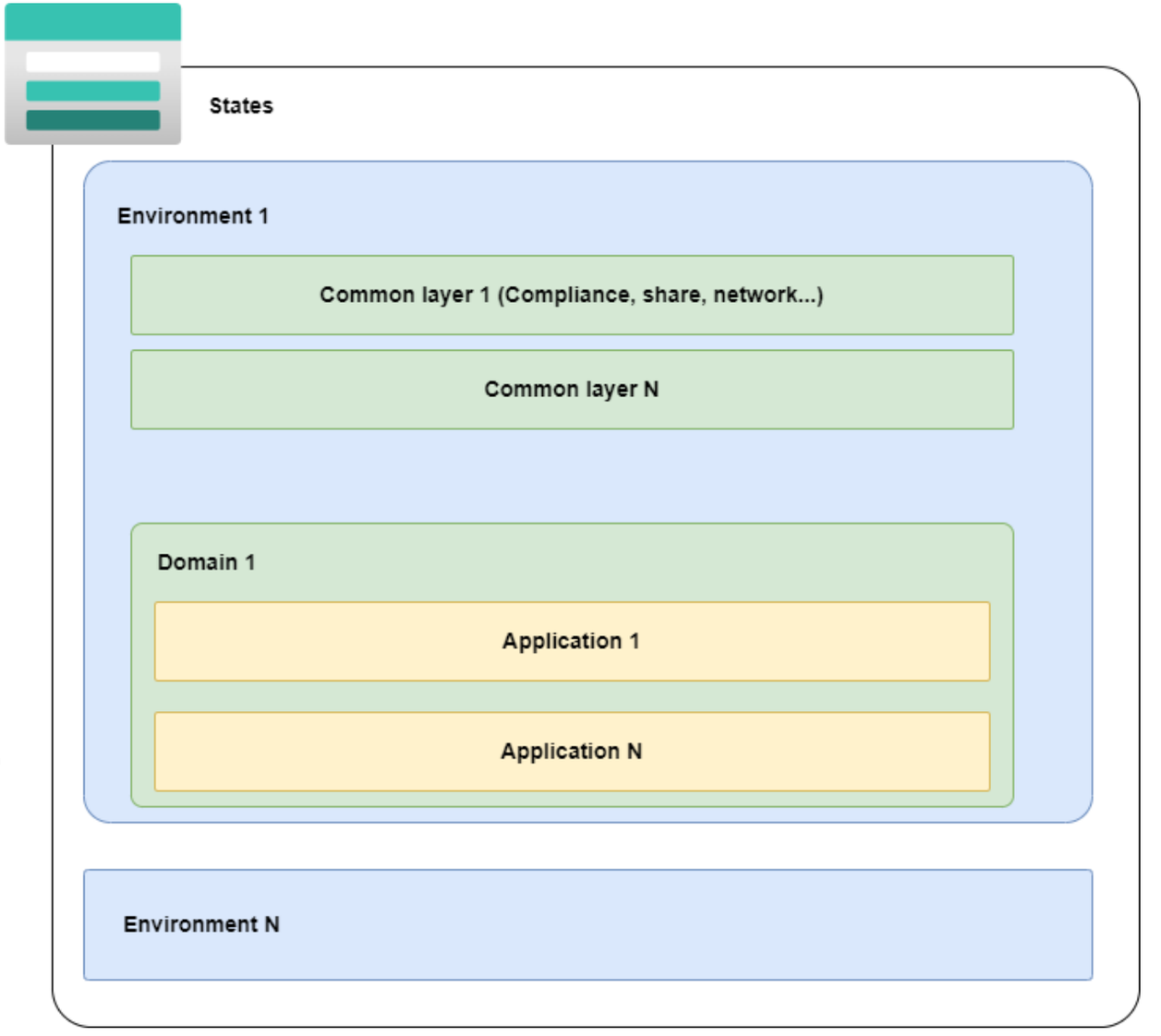
- The terraform workflows is always the same, try to mutualize it



# Structure the storage of your states

- Split your states **logically**
- Create a **structure convention** to store your states
- **Automatically predict** the path of your state
- For instance, for an application 1 the path to the state in the Storage Account can be:

/states/dev/domain1/application1/terraform.state



# Hands On Lab!

# Migrate to GitHub Actions

- Choose one project for instance the **prj1_infra_app** and add a .gitignore to it: https://www.toptal.com/developers/gitignore/api/terraform
- **Add any other files** like plan or .terraform* that are missing to the .gitignore if necessary
- Upload one project to GitHub
- Create a github **workflow** for the project
- Create a **job** to run a Terraform init, validate and plan
- Create another job to run a Terraform **apply** if changes where detected
- Use **terraform plan --detailed-exitcode**
- Use the **identity method** you prefer

# Tools

# Terraform-docs

- Generate documentation from Terraform modules in various output format
- https://github.com/terraform-docs/terraform-docs/
- One config file (.terraform-docs.yml) is used to set the template and the output of the documentation
- One simple command line to generate the documentation:

```
terraform-docs -c .terraform-docs.yml .
```

# TFLint

- terraform-linters/tflint: A Pluggable Terraform Linter (github.com)

- TFLint is a framework, and each feature is provided by plugins:
  - Find possible errors, like illegal instance types
  - Warn about deprecated syntax, unused declarations
  - Enforce best practices, naming conventions
  - Plugins for the Azure Rm provider is available: https://github.com/terraform-linters/tflint-ruleset-azurerm

- Use it **after** running terraform validate command

- A tflint Action is also available: https://github.com/terraform-linters/setup-tflint

# TFSec

- Made by Aqua Security
- https://aquasecurity.github.io/tfsec/
- Static analysis security scanner

# Checkov

- Made by bridgecrew (Prima Cloud)
- https://www.checkov.io/
- Checkov is a static code analysis tool
- Includes predefined policies to check for common misconfiguration issues
- Supports the creation and contribution of custom policies.
- Integrated with GitHub Actions: https://www.checkov.io/4.Integrations/GitHub%20Actions.html

# Terratest: End-2-end tests

- [Terratest | Automated tests for your infrastructure code. (gruntwork.io)]
- [Test Terraform modules in Azure using Terratest | Microsoft Docs]
- Tests are made using Go

**Write test code using Go**

Create a file ending in `_test.go` and run tests with the go test command. E.g., `go test my_test.go`.

**Use Terratest to deploy infrastructure**

Use Terratest to execute your real IaC tools (e.g., Terraform, Packer, etc.) to deploy real infrastructure (e.g., servers) in a real environment (e.g., AWS).

**Validate infrastructure with Terratest**

Use the tools built into Terratest to validate that the infrastructure works correctly in that environment by making HTTP requests, API calls, SSH connections, etc.

**Undeploy**

Undeploy everything at the end of the test.

# Infracost

- Infracost shows cloud cost estimates for Terraform.

- It lets engineers understand costs **before making changes**
  - In the **terminal**
  - In **VS Code**
  - In **pull requests**

- https://www.infracost.io/docs/

# Hands On Lab!

# Play with terraform-docs, TFLint and TFSec

- Use terraform-docs: https://terraform-docs.io/user-guide/introduction/
- Add a **.terraform-docs.yml** configuration file to at least one project.
- Try to generate a basic documentation in a file called **DOCS.md**
- Use TFLint: https://github.com/terraform-linters/tflint
- Add a .**tflint.hcl** configuration file to at least one project.
- Add the **azurerm** plugin
- Try to disable a **tflint rule**
- Use TFSec: https://github.com/aquasecurity/tfsec
- Run TFSec to at least one project

# Migrate to Terraform

# Migrate to Terraform

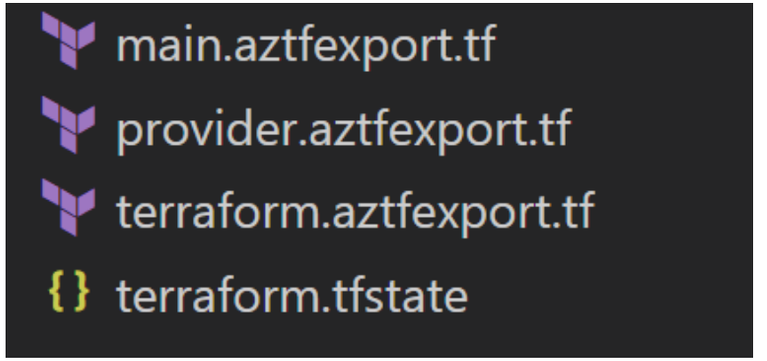
- There is **no perfect tools** to do this
- Multiple technics can be used:
- **Manually transform** the properties from ARM files to Terraform files
- **Manually transform** the properties from Bicep files to Terraform files
- Use **Microsoft Azure Export for Terraform** to speed up the process
- Use Terraform **import** command release with the version 1.5+
  https://www.hashicorp.com/blog/terraform-1-5-brings-config-driven-import-and-checks

# Microsoft Azure Export for Terraform

- https://github.com/Azure/aztfexport
- This will do the **big part** of the job but needs to be rework after
- It **extracts the configuration** from Azure and generate a Terraform code
- The Terraform configurations generated by **aztfexport** are **not meant to be comprehensive**
- It do **not ensure** that the infrastructure can be fully reproduced from the generated configurations.

# Microsoft Azure Export for Terraform

- This will save some time, but **you must double check** what was generated
- **Split it** correctly in multiple files and eventually in layers.
- This will generate 4 files:



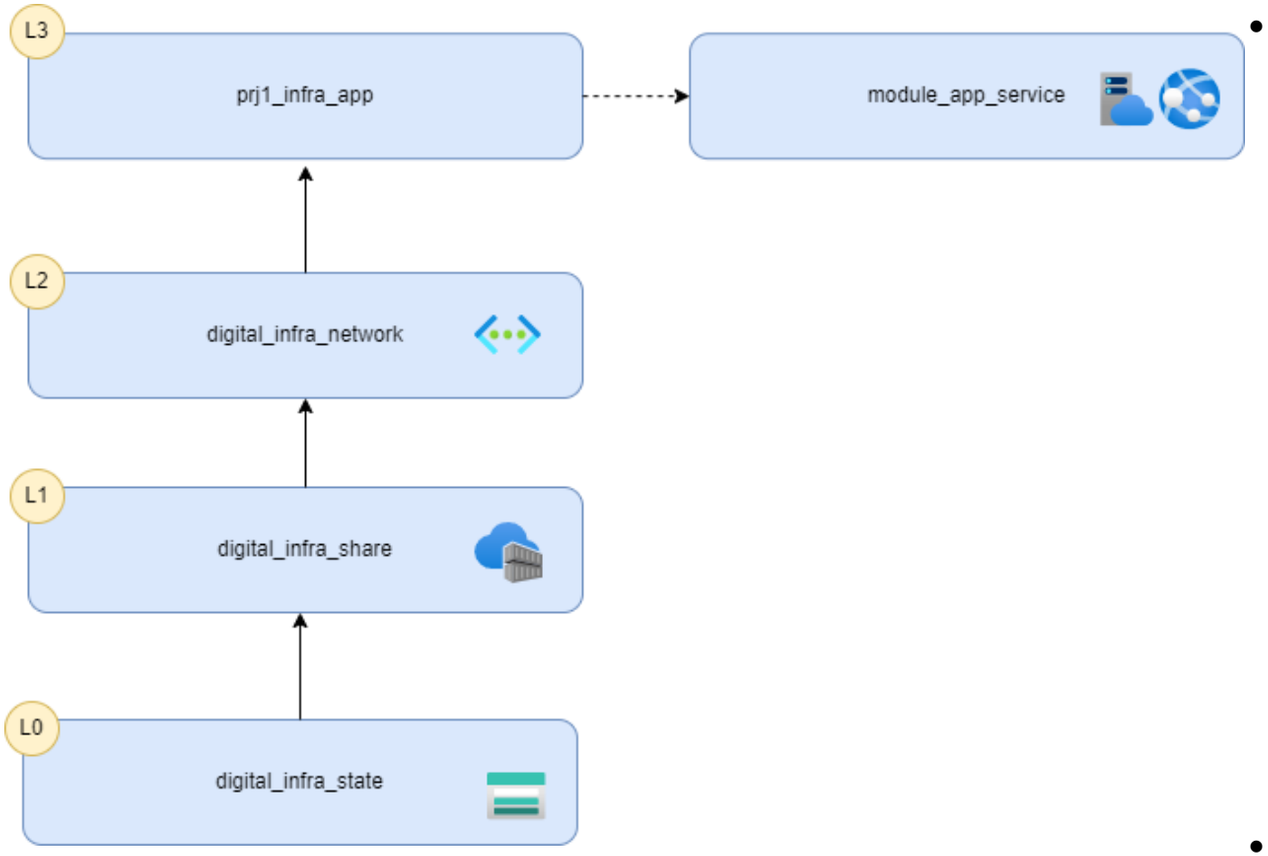

- All the infrastructure imported from Azure will be in one file: **main.aztfexport**

# Terraform import

- This command available since Terraform 1.5
- This generate all the code based on the resources you target
- Some default properties are generated but it can be a good starting point

# Hands On Lab!

# Migrate resource to Terraform




- Let's simulate the migration of a resource: Create an Azure Container Registry with Basic SKU in a resource group

- Then, try **one** of these options to do the Terraform code:

  Migrate ARM or Bicep template manually extracted to Terraform

  Try to generate it with **aztfexport** https://github.com/Azure/aztfexport

  Use Terraform **import** to generate it https://developer.hashicorp.com/terraform/tutorials/state/state-import#define-import-block

- Create a new folder called **digital_infra_share** and add this terraform code to it.

Microsoft Azure

# Thank you.