Terraform Training

Pre-requisites (Tools to install)

- Adobe Acrobat Reader or equivalent
- Docker Desktop
- Terraform
- Azure CLI or “Az” PowerShell module
- Visual Studio Code
- VS Code Plugins:
  - Azure Terraform
  - HashiCorp Terraform
  - JSON Formatter

Agenda

| Introduction to IaC

| Getting started with Terraform

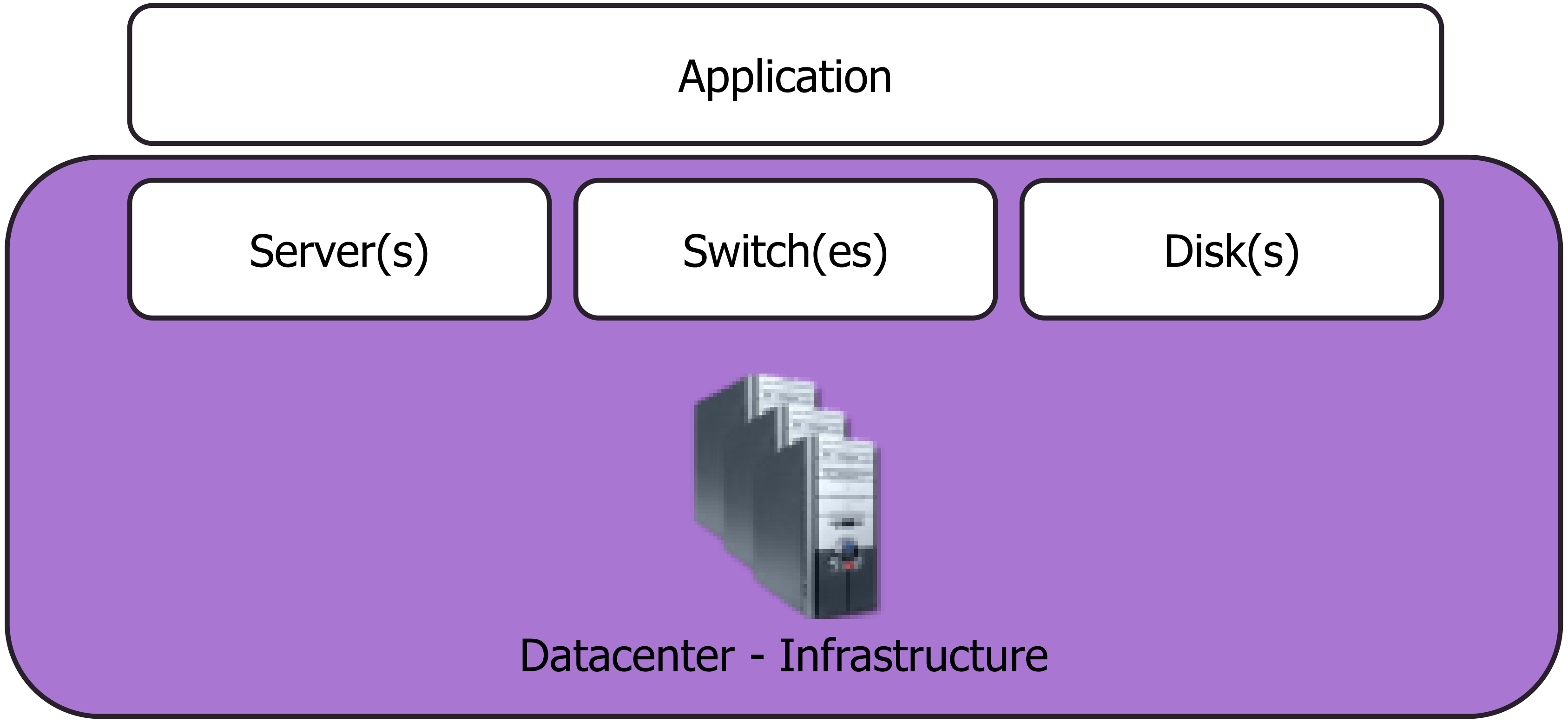
| Terraform in details

| Collaboration & pipeline

| Testing & deployment

1. Introduction to IaC

What is infrastructure ?



Application

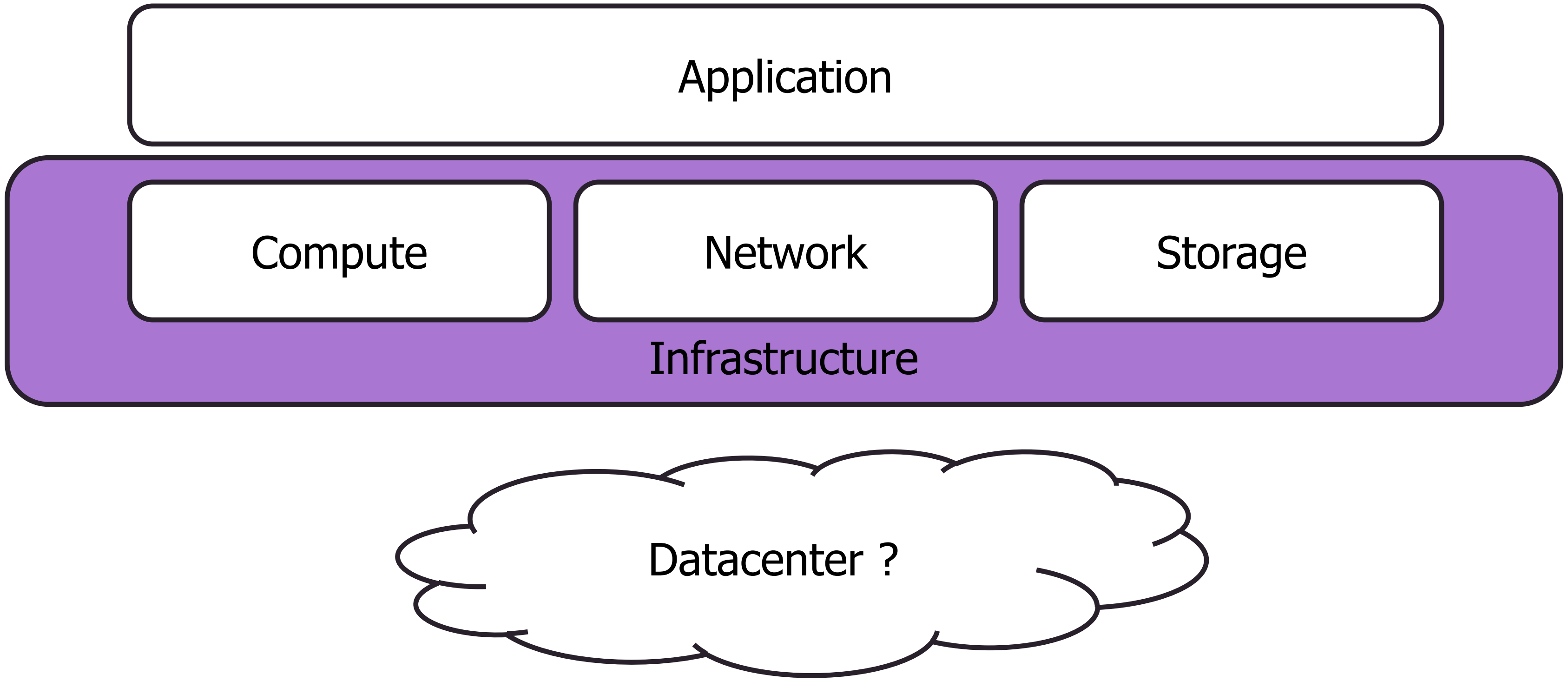
Server(s)

Switch(es)

Disk(s)

Datacenter - Infrastructure

What is infrastructure ?



Application

Compute

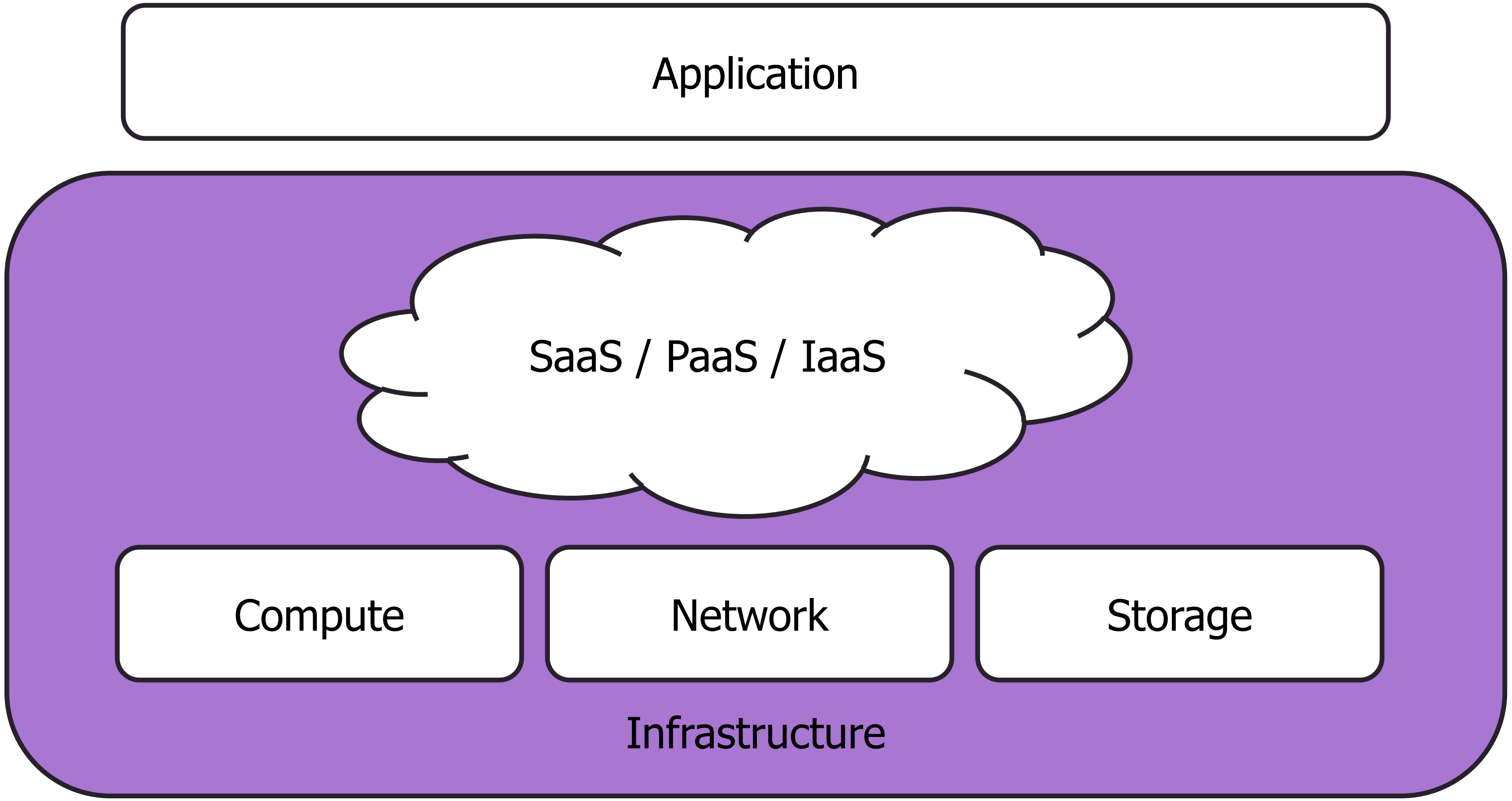
Network

Storage

Infrastructure

Datacenter ?

What is infrastructure ?


Application

SaaS / PaaS / IaaS

Compute

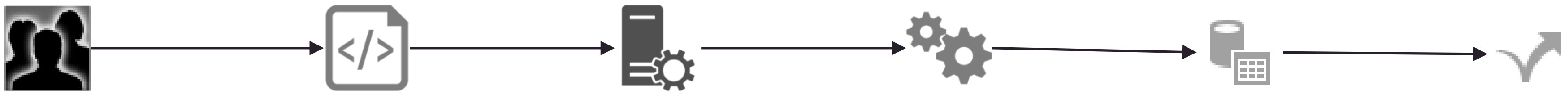
Network

Storage

Infrastructure

DevOps ?

- Culture
- **Automation**
- Measurement
- Sharing

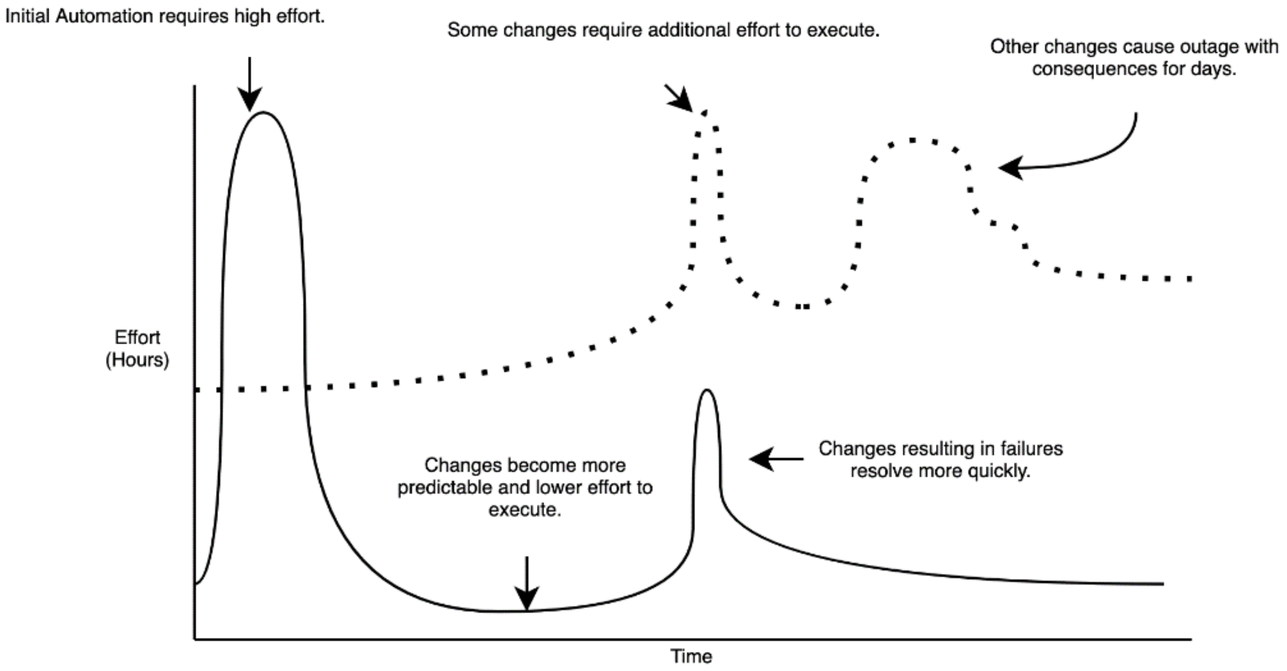


What is Infrastructure as Code (IaC) ?

- Definition
  - Process of automating infrastructure changes in a codified manner to achieve scalability, reliability, security and sustainability.
- Challenges
- Practices

Why use infrastructure as code ?

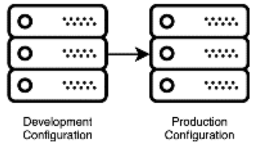
- Change management
- Return on time investment
- Knowledge sharing
- Security



Principles

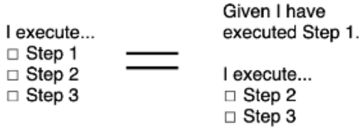
Reproducible

- Use a configuration to create a new environment with the same specification



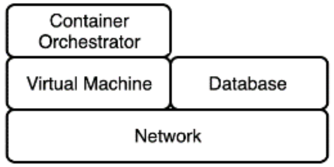
Idempotent

- Repeatedly run the automation on the same code and yield the same result



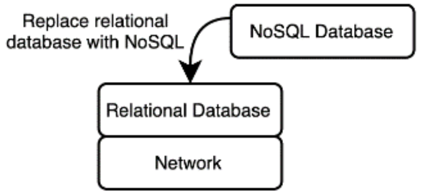
Composable

- Create an infrastructure system using a set of building blocks

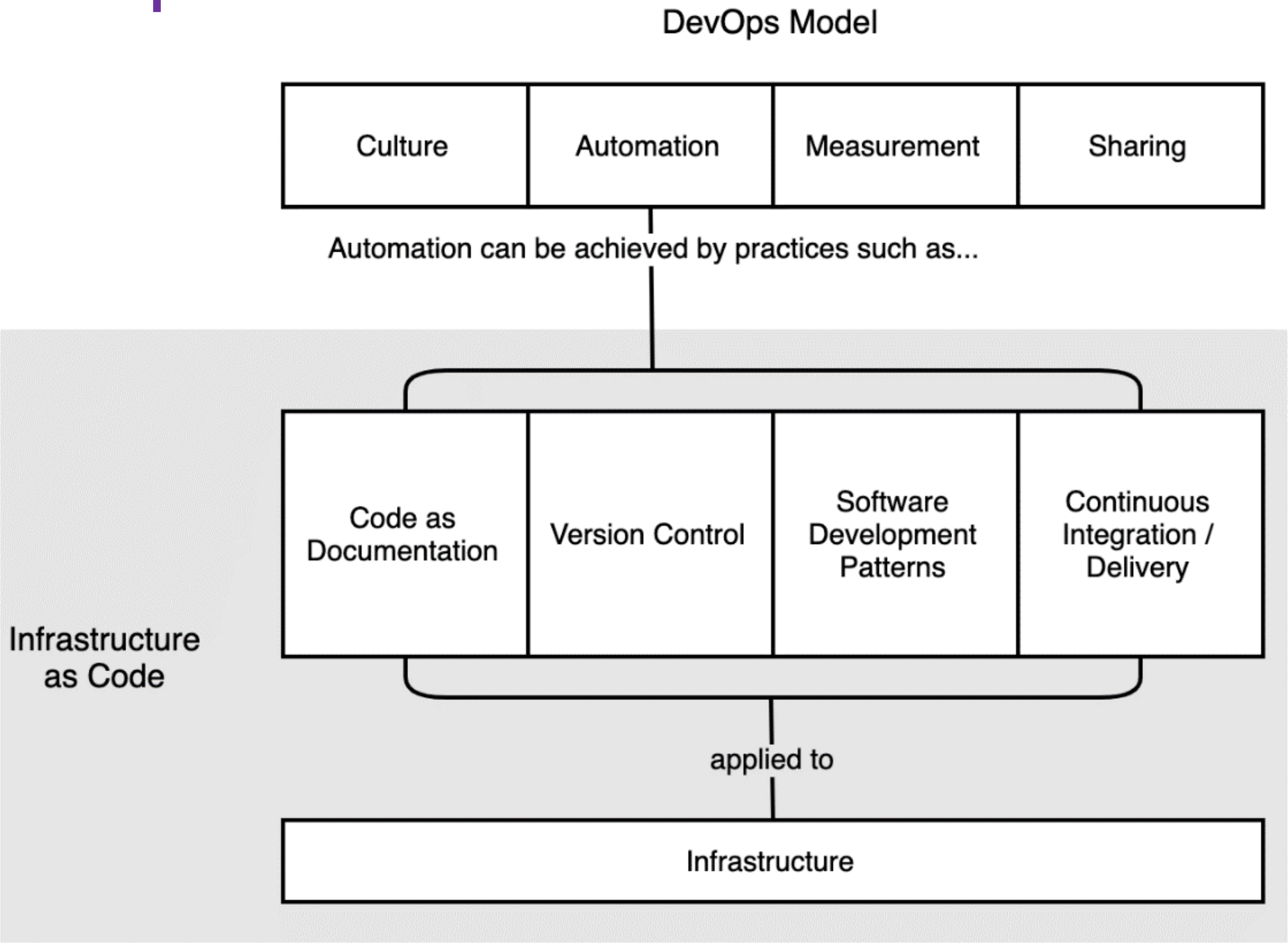


Evolvable

- Change part of the system with minimal disruption to other infrastructure



Quick recap



DevOps Model

Culture | Automation | Measurement | Sharing

Automation can be achieved by practices such as...

Code as Documentation | Version Control | Software Development Patterns | Continuous Integration / Delivery

Infrastructure as Code

applied to

Infrastructure

Tools

Provisioning

Tool	Provider
Azure Resource Manager	Microsoft Azure
HashiCorp Terraform	Various
Pulumi SDK	Various
AWS Cloud Development Kit	Amazon Web Services
Kubernetes Manifests	Kubernetes (Container Orchestrator)

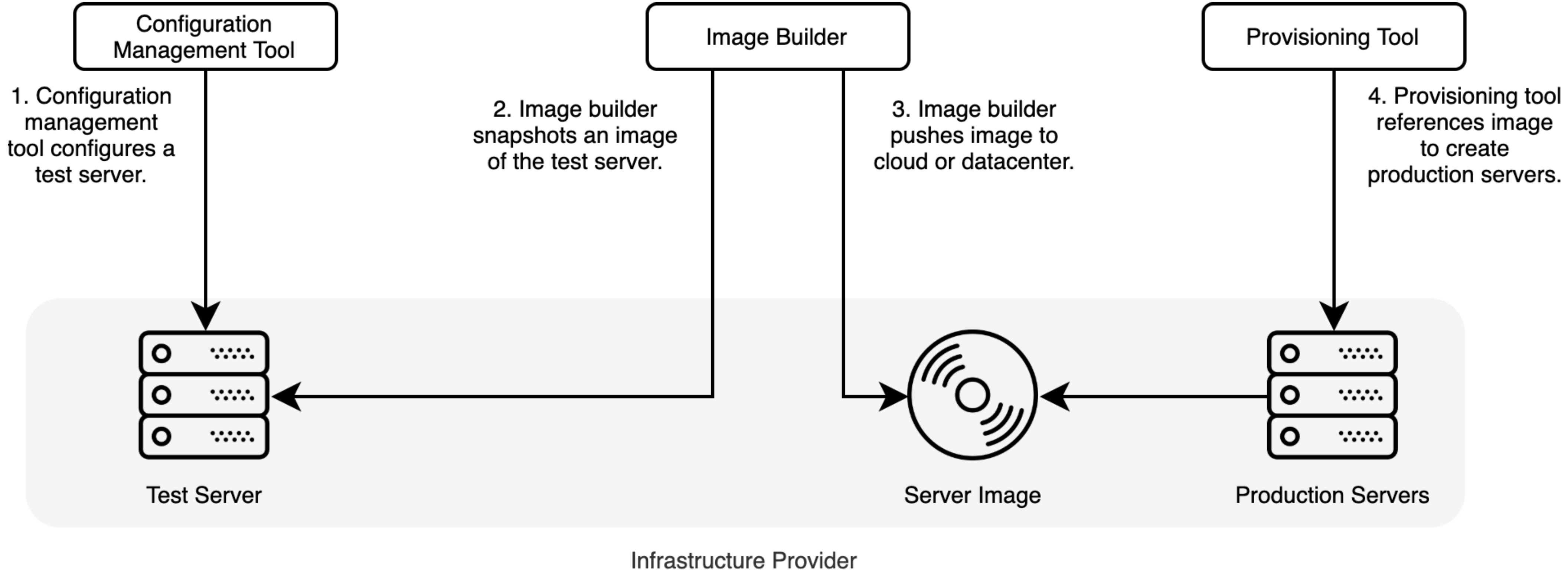
Image building

Tool	Runtime Environment	Build Target
HashiCorp Packer	Containers & Servers	Various
Docker	Containers	Container Registries
Amazon EC2 Image Builder	Servers	Amazon Web Services
Azure VM Image Builder	Servers	Microsoft Azure

Configuration Management

- Chef, Puppet, Ansible, SaltStack, CFEngine, …

Deployment process in a nutshell



Configuration Management Tool

Image Builder

Provisioning Tool

1. Configuration management tool configures a test server.

2. Image builder snapshots an image of the test server.

3. Image builder pushes image to cloud or datacenter.

4. Provisioning tool references image to create production servers.

Quiz - Solution

- Infrastructure can be software, platform, or hardware that delivers or deploys applications to production.
- Infrastructure as code is a DevOps practice of automating infrastructure to achieve reliability, scalability, and security.
- The principles of infrastructure as code are reproducibility, idempotency, composability, and evolvability.
- By following the principles of infrastructure as code, you can improve change management processes, lower time spent on fixing failed systems in the long term, better share knowledge and context, and build security into your infrastructure.
- Infrastructure as code tools include three types: provisioning tools, configuration management tools, and image builders.

IaC - Summary

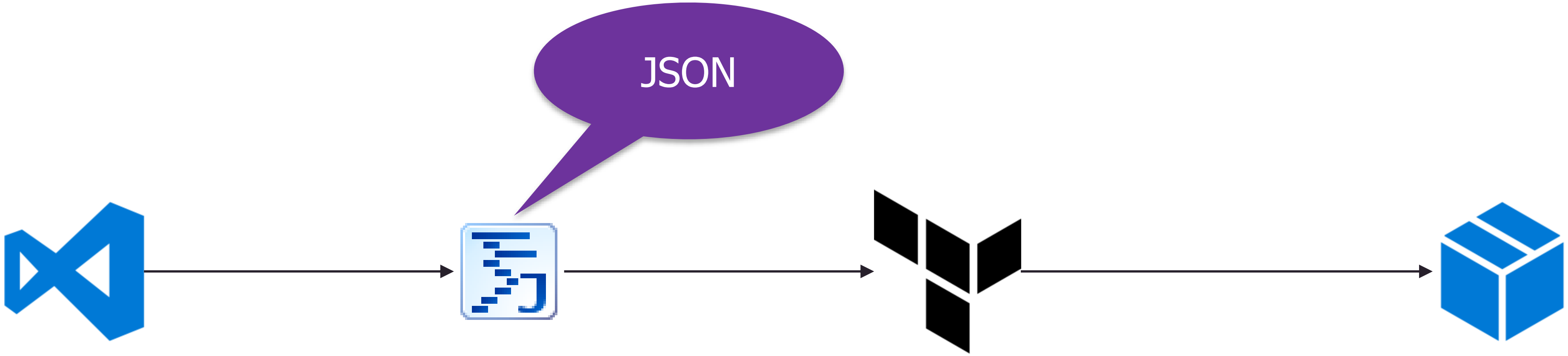
- Infrastructure can be software, platform, or hardware that delivers or deploys applications to production.
- Infrastructure as code is a DevOps practice of automating infrastructure to achieve reliability, scalability, and security.
- The principles of infrastructure as code are reproducibility, idempotency, composability, and evolvability.
- By following the principles of infrastructure as code, you can improve change management processes, lower time spent on fixing failed systems in the long term, better share knowledge and context, and build security into your infrastructure.
- Infrastructure as code tools include three types: provisioning tools, configuration management tools, and image builders.

2. Getting Started with Terraform

Content

- Generating and applying execution plans
- Analysing when function hooks are triggered by Terraform
- Utilizing the Local provider to create and manage files
- Simulating, detecting, and correcting for configuration drift
- Understanding the basics of Terraform state management

In a nutshell



JSON

Authoring Tool

Code/Script with
.tf extension

Terraform references
any .tf file

Terraform creates, updates
or deletes infrastructure resources

JSON ?

- What is it ?
  - JSON stands for JavaScript Object Notation
  - JSON is a text format for storing and transporting data
- Why do we need it ?
  - JSON is "self-describing" and easy to understand
  - JSON is a lightweight data-interchange format
  - JSON is language independent

JSON Syntax

- JSON syntax is derived from JavaScript object notation syntax:
    - Data is in name/value pairs
    - Data is separated by commas
    - Curly braces hold objects
    - Square brackets hold arrays

```
{"employees":[
        { "firstName":"John", "lastName":"Doe" },
        { "firstName":"Anna", "lastName":"Smith" },
        { "firstName":"Peter", "lastName":"Jones" }
]}
```

JSON Example

- A site represents a set of buildings
- A building has
  - An address/location
  - An identifier
  - A name
  - A collection of rooms
- A building is assigned to a department
- A room has a name and a max number of persons it can host

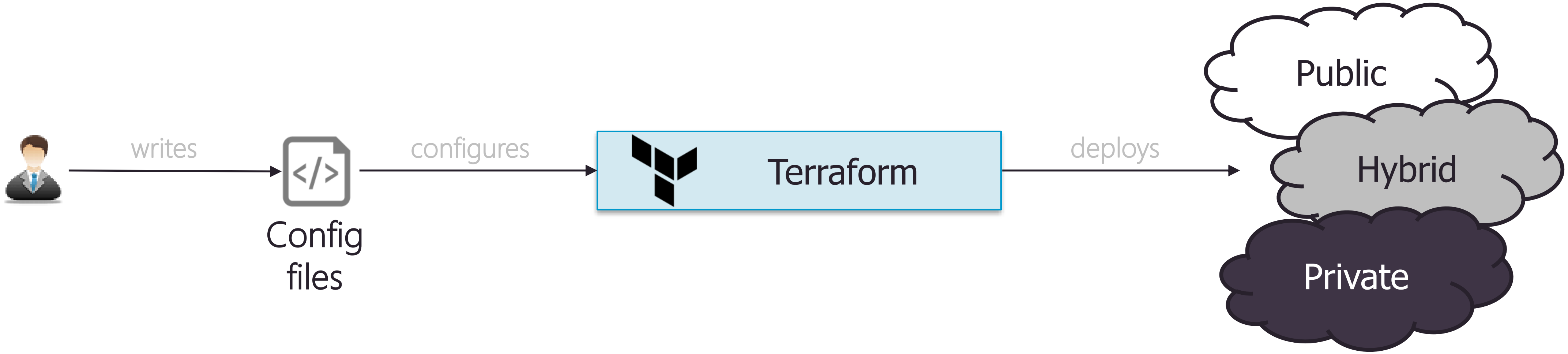
```
{
"site": [
   "B20": {
        "address": "Nowhere street, 20",
        "identifier": 123456798,
        "name": "building-20",
        "member-of": [
                "IT",
                "Support"
        ],
        "rooms": [
                "room#1": {
                        "name":"R2.2.1",
                        "capacity": 4
                },
                "room#2": {
                        "name":"R2.2.1",
                        "capacity": 4
                }
        ]
   },
   "B21": {}
]}
```

Exercise 2.01 – JSON File

Check README file

Terraform ?

Terraform is a deployment technology to provision and manage IaC.

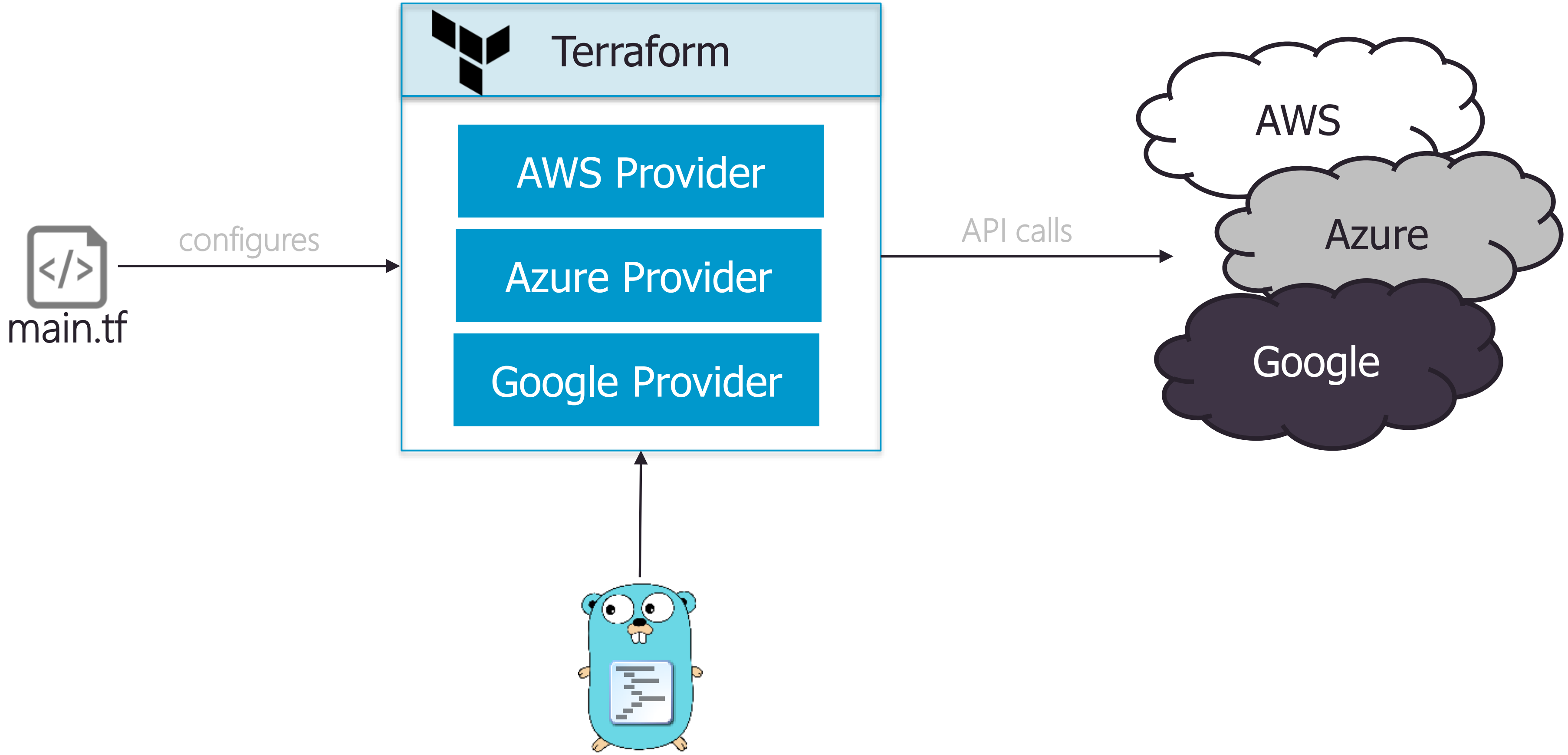


Terraform ?

- The provisioning and management is done through the execution of commands on
  - Workspace
  - State
  - Infrastructure

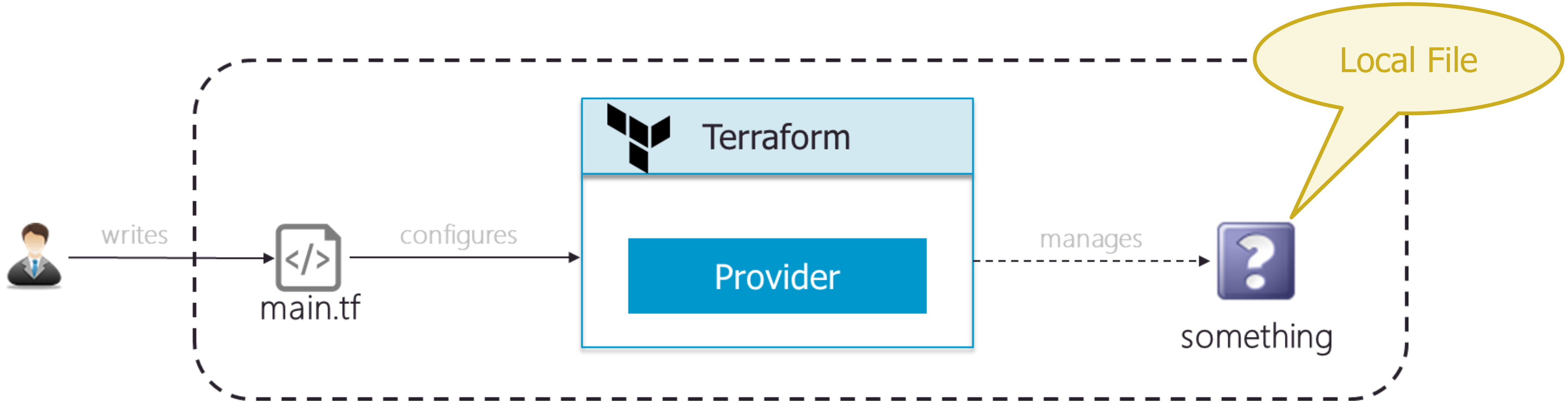
- An Terraform infrastructure is represented by
  - A set of resources [1..n]
  - A set of providers [1..n]
  - A set of data [0..n]

Terraform ?


Terraform
AWS Provider
Azure Provider
Google Provider
main.tf
configures
API calls
AWS
Azure
Google

Note: Local-only resources

- There are different sorts of resources: local-only resources.
  - Examples ?



Why Terraform ?

- Key characteristics:
  - **Provisioning tool**: Deploys infrastructure, not just applications
  - **Easy to use**: For most of us (ie: non geniuses)
  - **Free and Open Source**: Who doesn’t like free?
  - **Declarative**: Say what you want, not how to do it
  - **Cloud agnostic**: Deploy to any cloud using the same tool
  - **Expressive and extendable**

Terraform vs other IaC tools

	Provisioning tool	Easy to use	Free and Open Source	Declarative	Cloud Agnostic	Expressive and extendable
Ansible		X	X		X	X
Chef			X	X	X	X
Puppet			X	X	X	X
SaltStack		X	X	X	X	X
Terraform	X	X	X	X	X	X
Pulumi	X		X		X	X
AWS CloudFormation	X	X		X		
GCP Deployment Manager	X	X		X		
Azure Resource Manager	X			X		

"Hello World !"

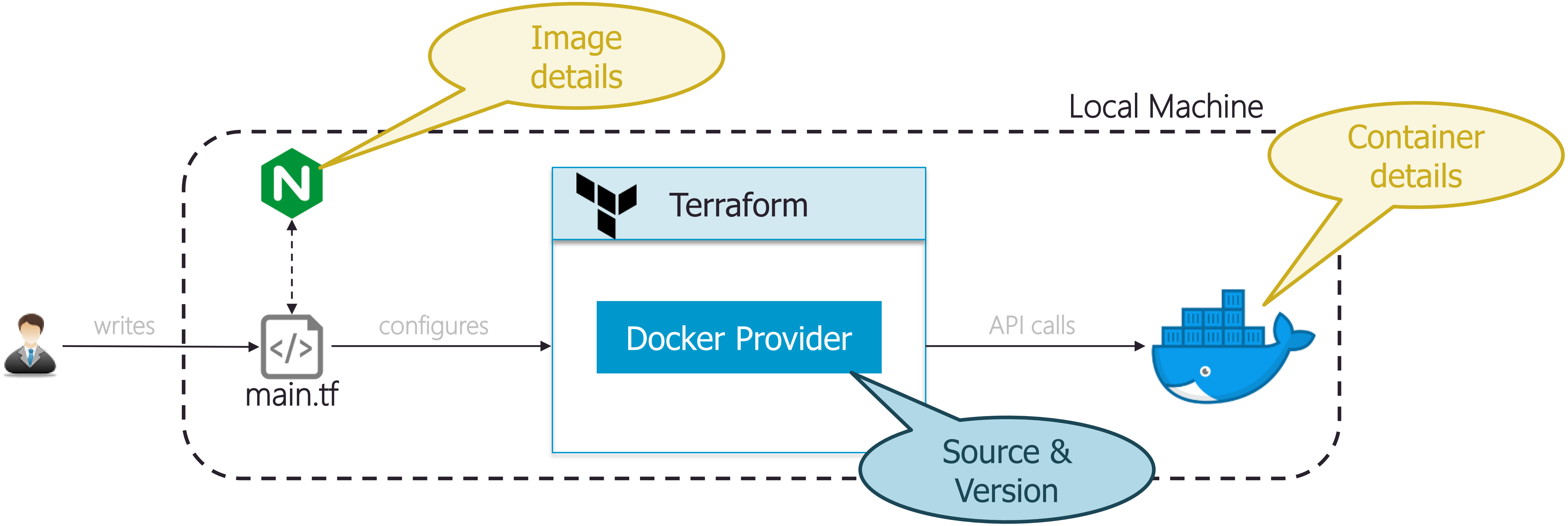
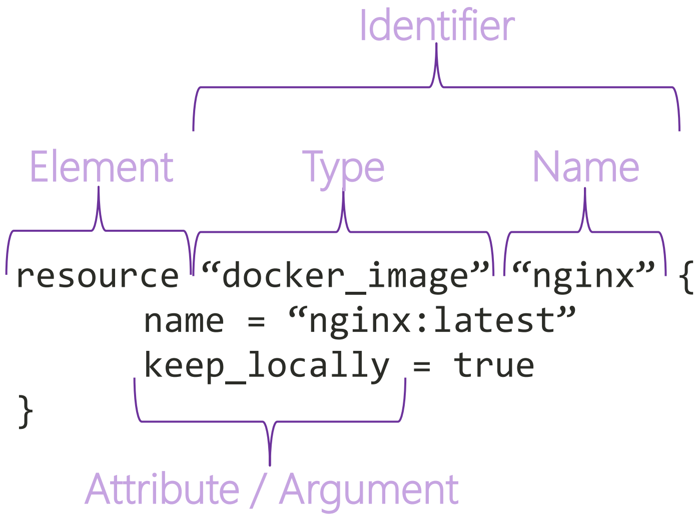


Image details
Local Machine
Container details
Terraform
writes
configures
API calls
Docker Provider
main.tf
Source & Version

Terraform resource

- Image
  - Name: nginx
  - Keep_locally: true



```
resource “docker_image” “nginx” {
          name = “nginx:latest”
          keep_locally = true
}
```

Terraform provider

- Provider: docker
    - Source: kreuzwerker/docker
    - Version: ~> 2.13.0

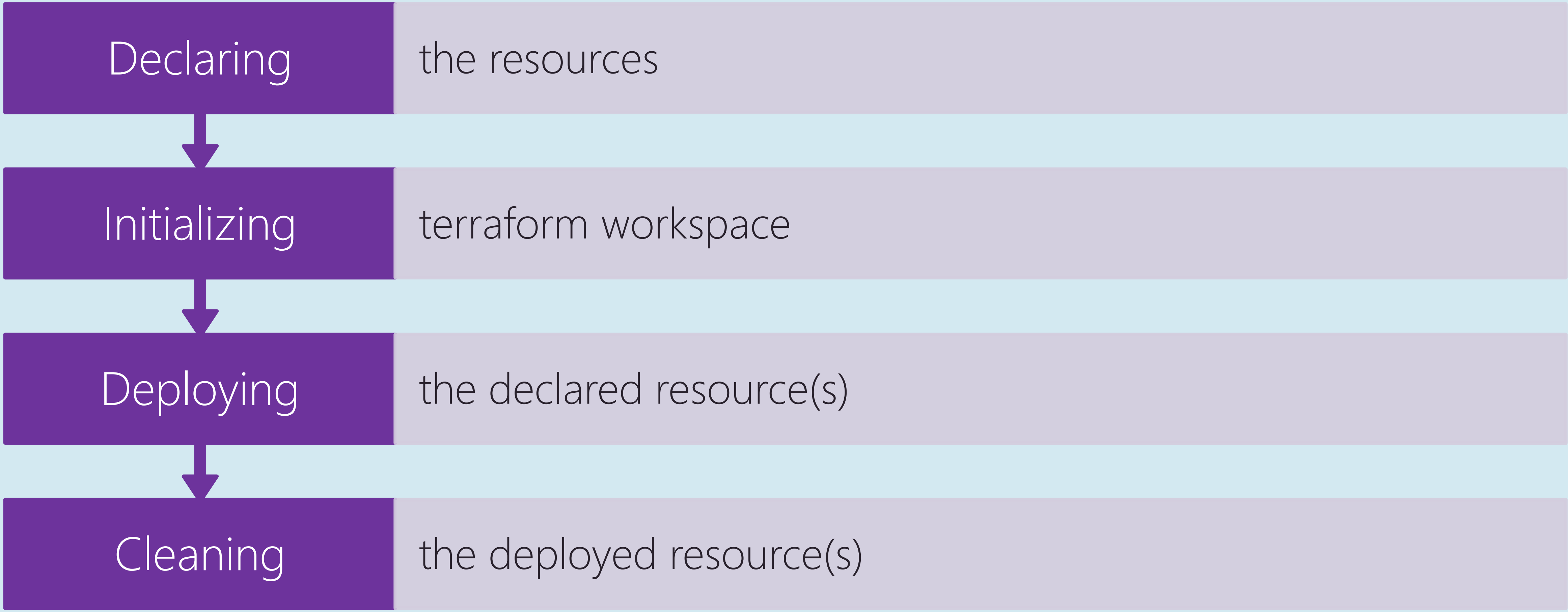
```
terraform {
        required_providers {
                docker = {
                        source = “kreuzwerker/docker”
                        version = “~> 2.13.0”
                }
        }
}
```

Exercise 2.02 – Docker – Hello world!

Check README file

- Terraform
  - Provider
    - Source: kreuzwerker/docker
    - Version: ~> 2.13.0
- Docker resources
  - Image
  - Container:
    - Image: docker_image.nginx.latest
    - Name: helloW
    - Ports:
      - Internal: 80
      - External: 8000

Deployment process in a nutshell (demo)



Declaring the resources

Initializing terraform workspace

Deploying the declared resource(s)

Cleaning the deployed resource(s)

Exercise 2.03 – Azure – Hello world!

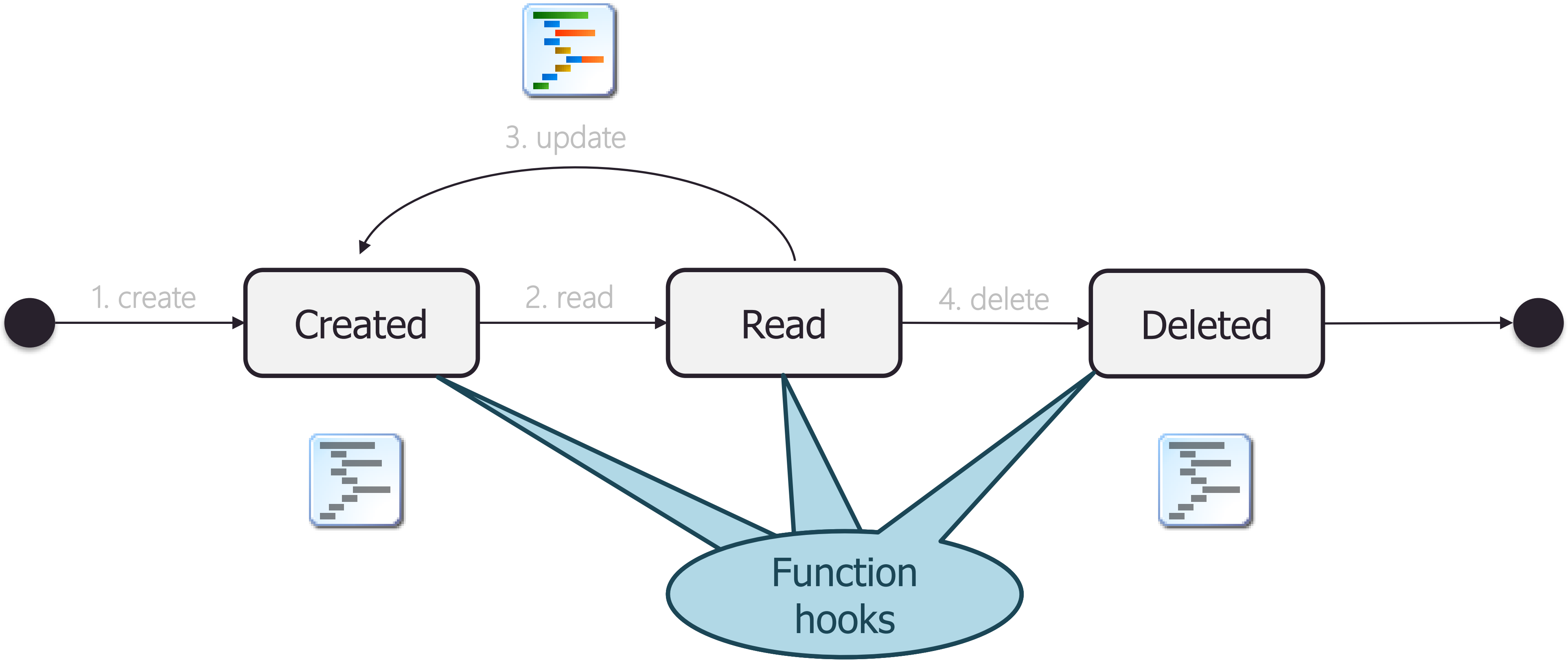
Check README file

> Assumption: familiar with Azure concepts & particularly resource groups

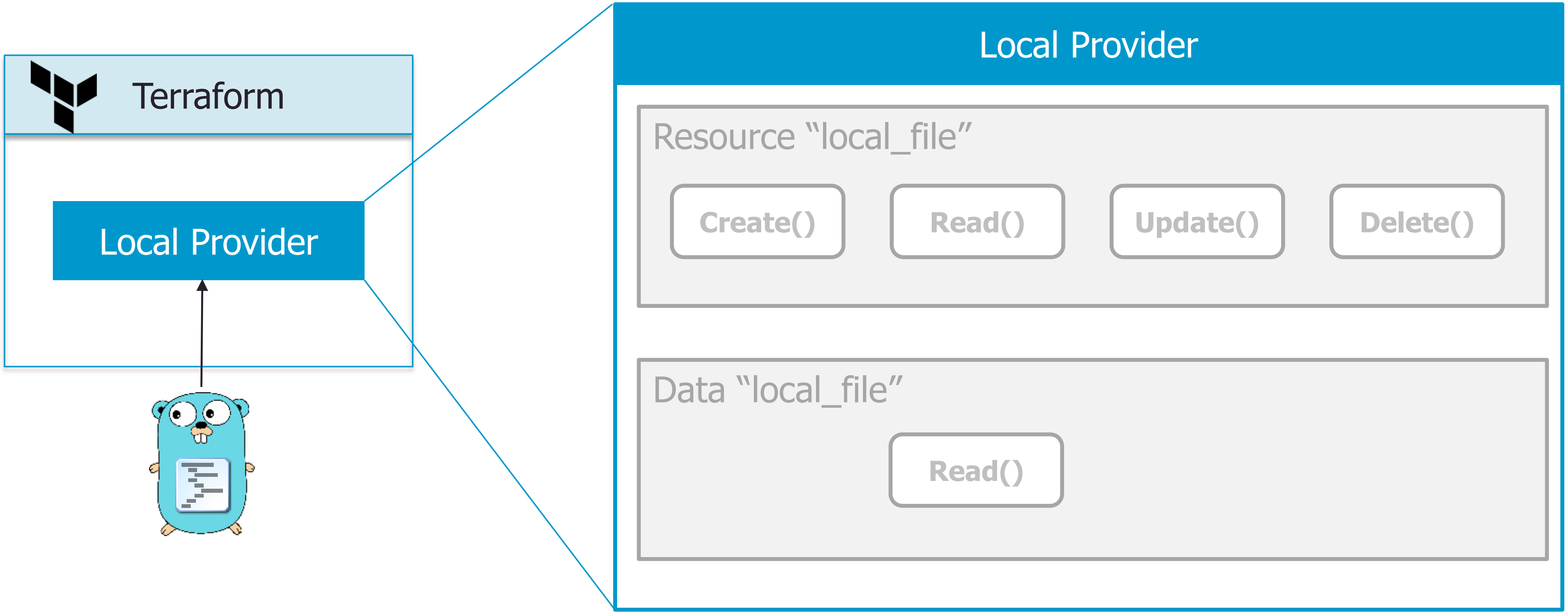
So far

- Terraform is a declarative, provisioning state management tool that performs CRUD operations on managed resources deployed onto any public or private cloud.
- The major elements of Terraform are resources, data sources and providers.
- To deploy a Terraform project you must first write configuration code, then configure providers and other input variables, initialize Terraform and finally apply changes. Clean-up is done with a destroy run.

Resource lifecycle (States)



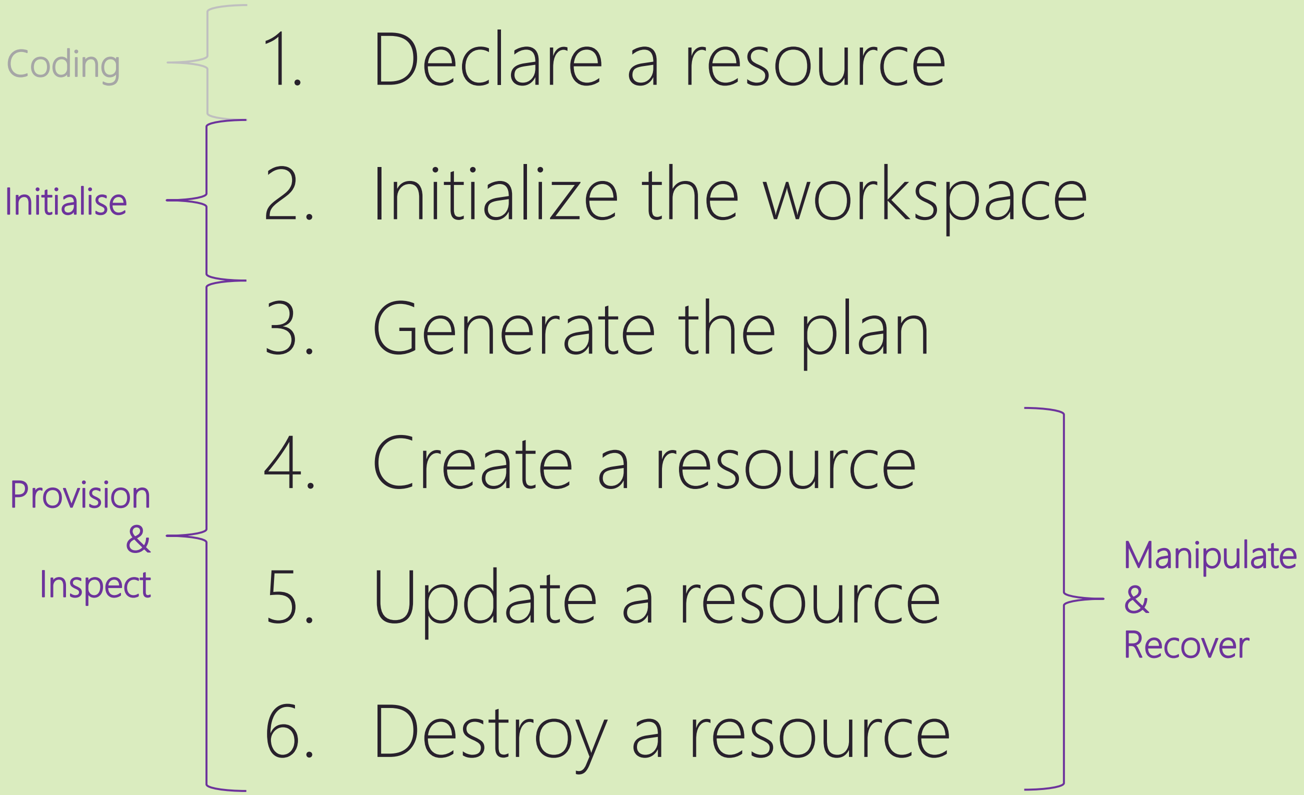
Note: Function hook



Terraform commands

- Infrastructure (data, resource & provider)
  - Initialise: `init`, `get`
  - Provision: `plan`, `apply`, `destroy`
  - Inspect: `graph`, `output`, `show`
  - Import: `import`
- State
  - Manipulate: `list`, `show`, `refresh`, …
  - Recover: `pull`, `push`, `force-unlock`, …
- Workspace
  - Manage: `list`, `select`, `new`, `delete`, `show`

Example 2.04 - Lifecycle



Coding

1. Declare a resource

Initialise

2. Initialize the workspace

Provision & Inspect

3. Generate the plan
4. Create a resource
5. Update a resource
6. Destroy a resource

Manipulate & Recover

Declare a local file resource

Block 1

```
terraform {
required_version = ">= 0.15"
required_providers {
local = {
source = "hashicorp/local"
version = "~> 2.0"
}
}
}
```

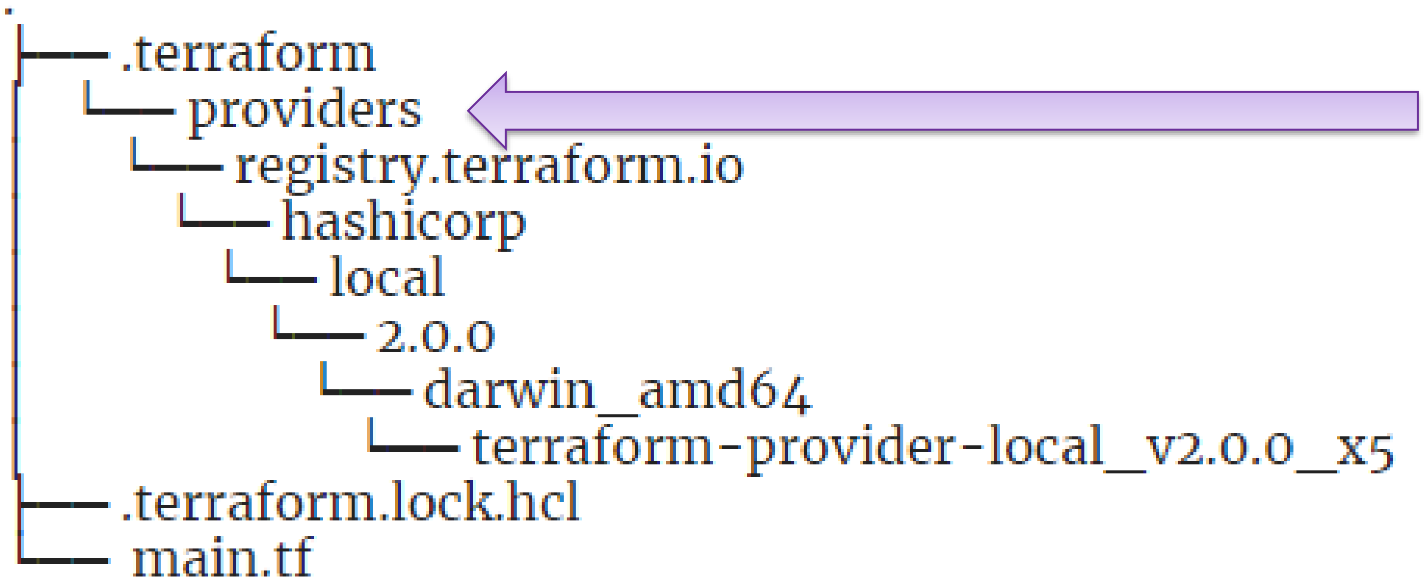
Block 2

```
resource "local_file" "Britney" {
filename = "I_did_it_again.txt"
content = <<-EOF
'Cause to lose all my senses

That is just so typically me
You see, my problem is this, I'm dreaming away
Can't you see I'm a fool in so many ways?
EOF
}
```

HEREDOC SYNTAX

Initializing the workspace

```
terraform init
```



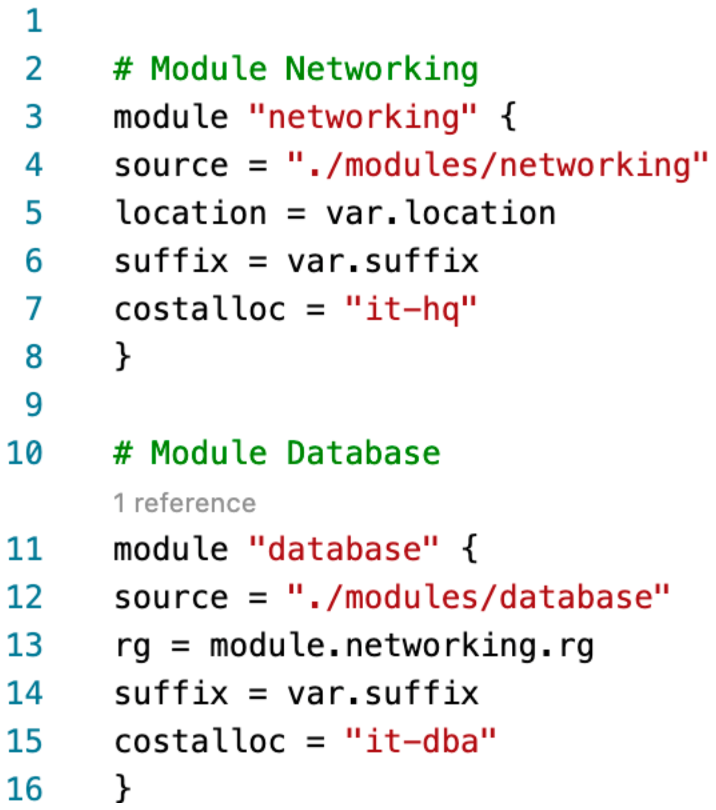
```
.
├── .terraform
│   └── providers
│       └── registry.terraform.io
│           └── hashicorp
│               └── local
│                   └── 2.0.0
│                       └── darwin_amd64
│                           └── terraform-provider-local_v2.0.0_x5
├── .terraform.lock.hcl
└── main.tf
```

VERSION

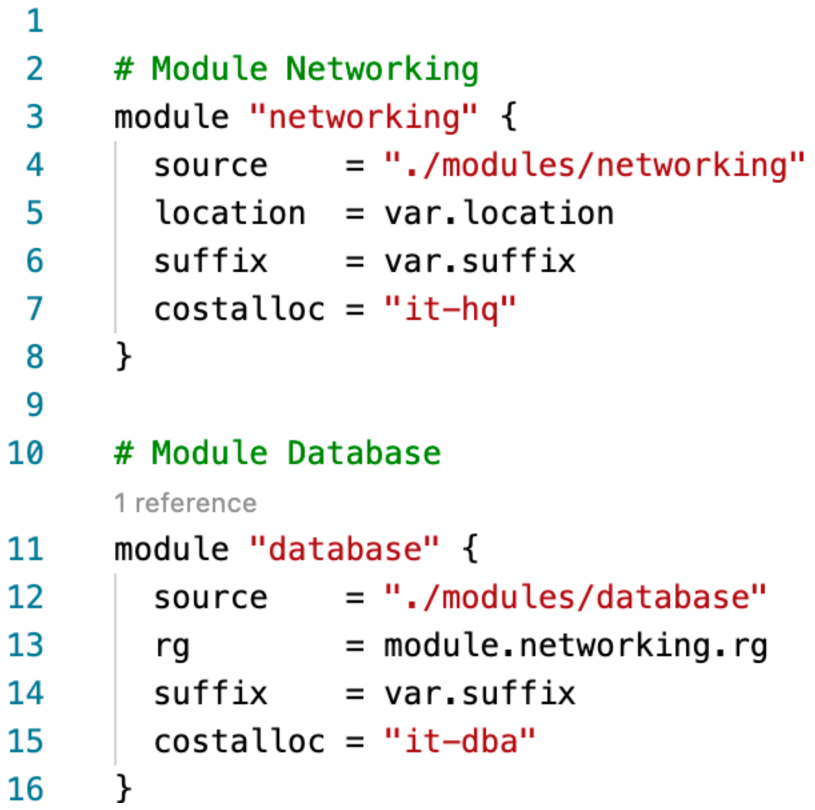
Format the script

```
terraform fmt -recursive
```

```
terraform fmt -write=false -diff=true
```



```

# Module Networking
module "networking" {
source = "./modules/networking"
location = var.location
suffix = var.suffix
costalloc = "it-hq"
}

# Module Database
1 reference
module "database" {
source = "./modules/database"
rg = module.networking.rg
suffix = var.suffix
costalloc = "it-dba"
}
```



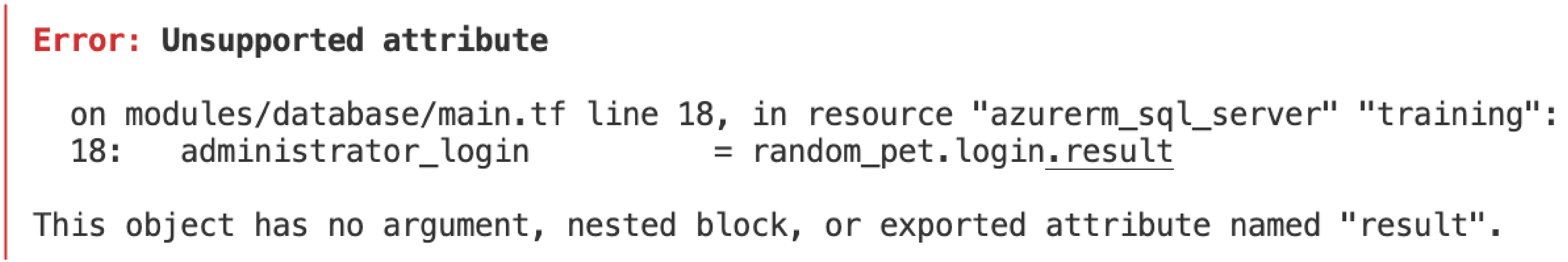
```

# Module Networking
module "networking" {
  source    = "./modules/networking"
  location  = var.location
  suffix    = var.suffix
  costalloc = "it-hq"
}

# Module Database
1 reference
module "database" {
  source    = "./modules/database"
  rg        = module.networking.rg
  suffix    = var.suffix
  costalloc = "it-dba"
}
```

Validate the script

```
terraform validate
```

or

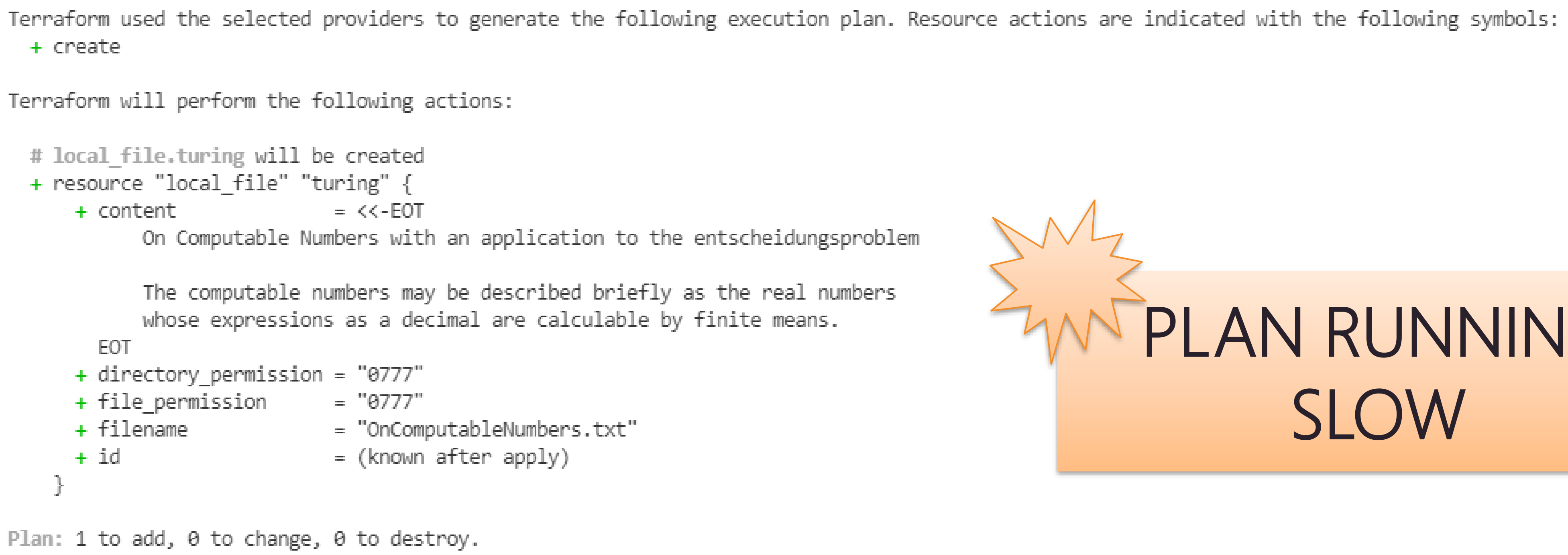
```
terraform validate -json
```



```
Error: Unsupported attribute

  on modules/database/main.tf line 18, in resource "azurerm_sql_server" "training":
  18:   administrator_login          = random_pet.login.result

This object has no argument, nested block, or exported attribute named "result".
```

Generating the execution plan

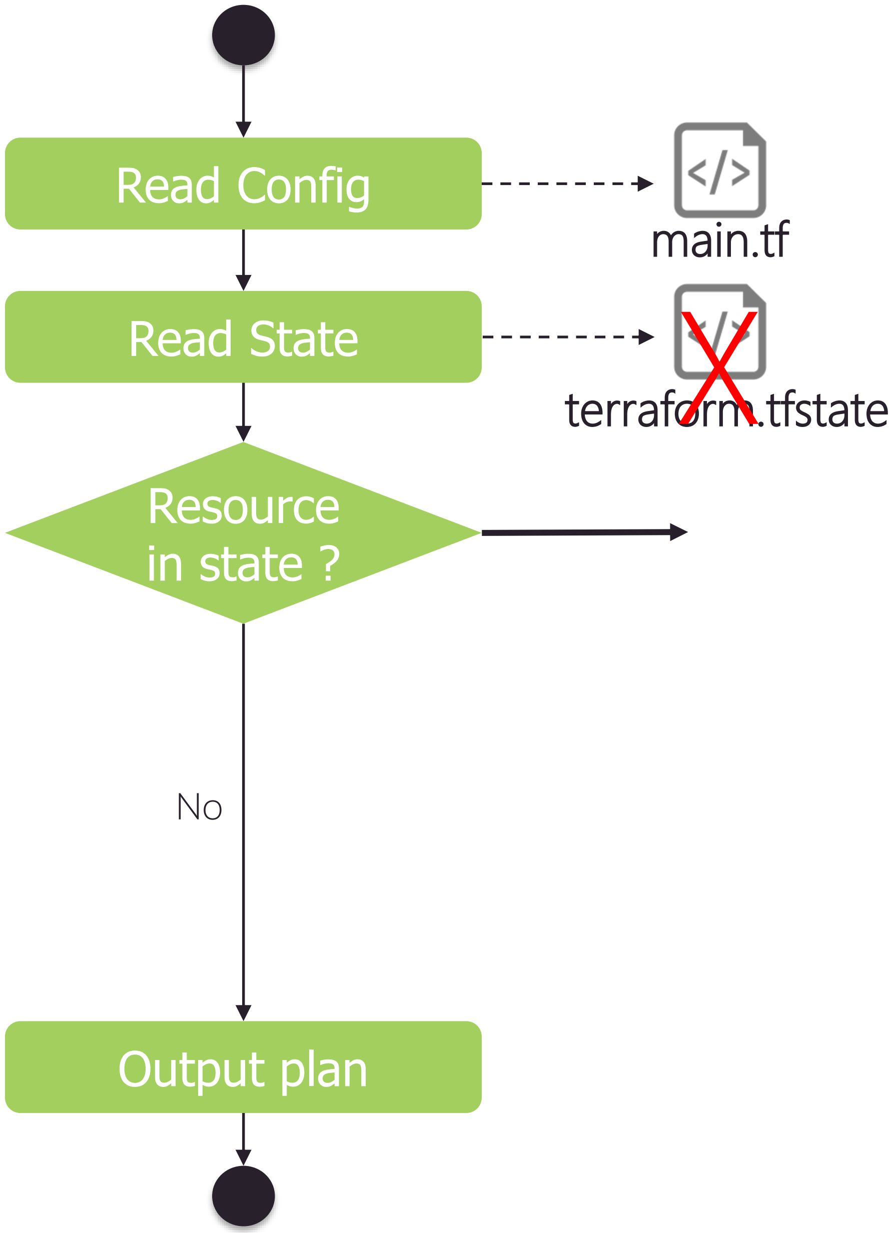
- Plan is a static code analysis, read-only, idempotent and “*robust*”
- Always run a terraform plan before deploying



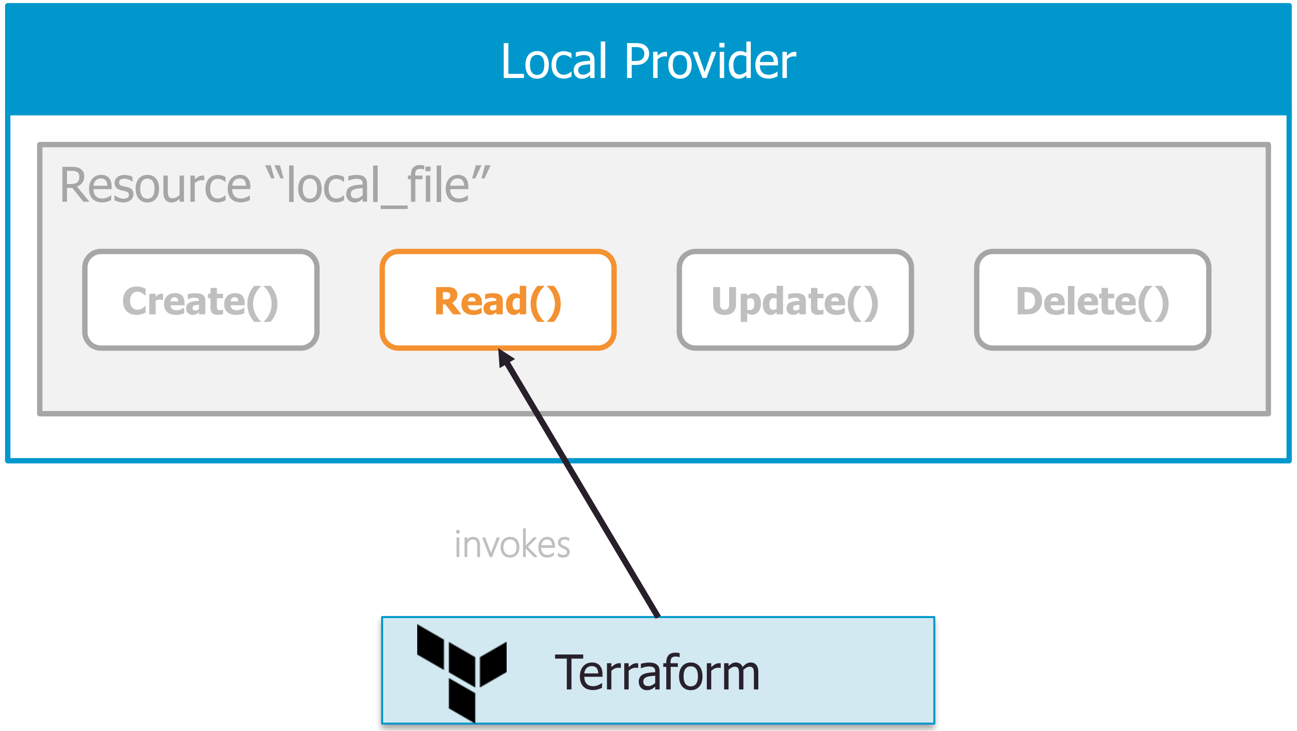
```
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # local_file.turing will be created
  + resource "local_file" "turing" {
      + content              = <<-EOT
            On Computable Numbers with an application to the entscheidungsproblem

            The computable numbers may be described briefly as the real numbers
            whose expressions as a decimal are calculable by finite means.
        EOT
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "OnComputableNumbers.txt"
      + id                   = (known after apply)
    }

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

PLAN RUNNING SLOW

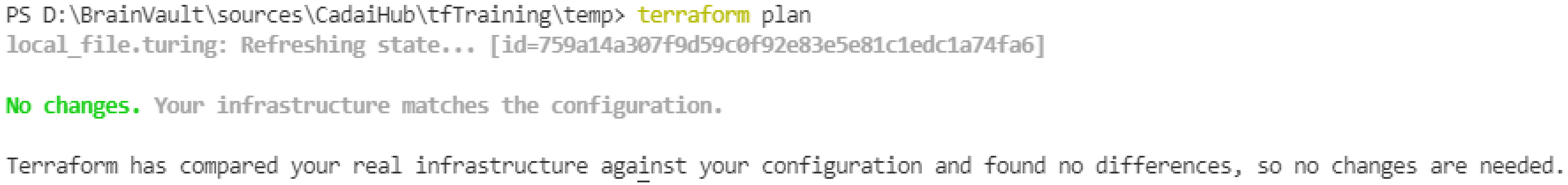
So far


Read Config
main.tf
Read State
terraform.tfstate
Resource in state ?
No
Output plan

No-op()



read.go



```
PS D:\BrainVault\sources\CadaiHub\tfTraining\temp> terraform plan
local_file.turing: Refreshing state... [id=759a14a307f9d59c0f92e83e5e81c1edc1a74fa6]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.
```

Generating the execution plan

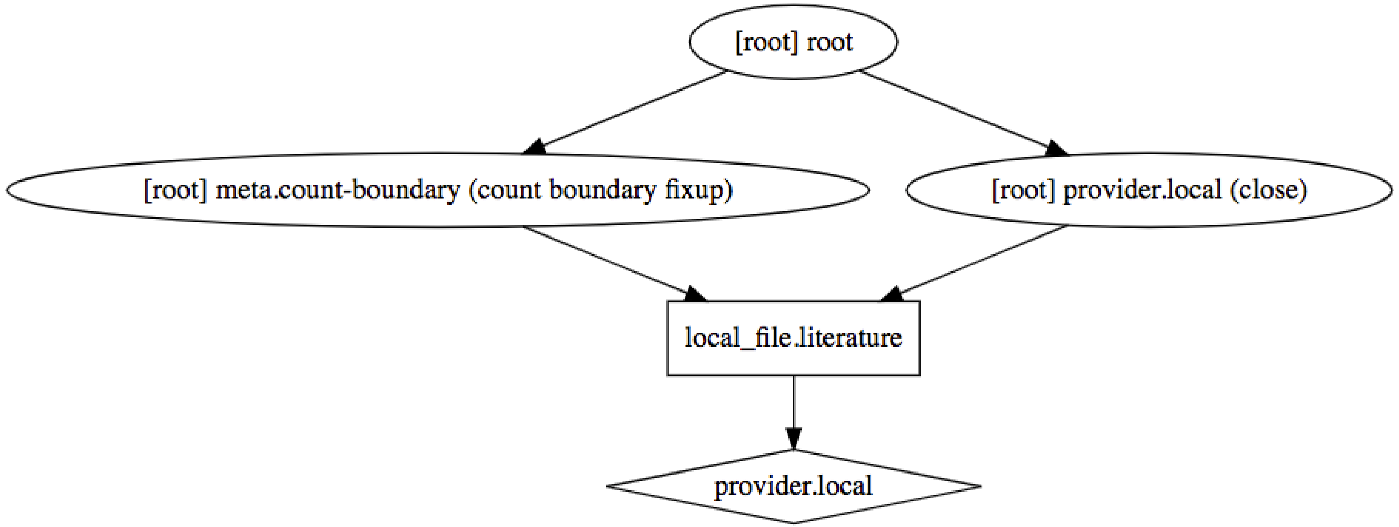
```
terraform plan -out plan.out
```

```
terraform show -json plan.out > plan.json
```

TRACING

Visualizing the plan

```
digraph {
        compound = "true"
        newrank = "true"
        subgraph "root" {
                "[root] local_file.turing (expand)" [label = "local_file.turing", shape = "box"]
                "[root] provider[\"registry.terraform.io/hashicorp/local\"]" [label = "provider[\"registry.terraform.io/hashicorp/local\"]", shape = "diamond"]
                "[root] local_file.turing (expand)" -> "[root] provider[\"registry.terraform.io/hashicorp/local\"]"
                "[root] meta.count-boundary (EachMode fixup)" -> "[root] local_file.turing (expand)"
                "[root] provider[\"registry.terraform.io/hashicorp/local\"] (close)" -> "[root] local_file.turing (expand)"
                "[root] root" -> "[root] meta.count-boundary (EachMode fixup)"
                "[root] root" -> "[root] provider[\"registry.terraform.io/hashicorp/local\"] (close)"
        }
}
```


[root] root
[root] meta.count-boundary (count boundary fixup)
[root] provider.local (close)
local_file.literature
provider.local

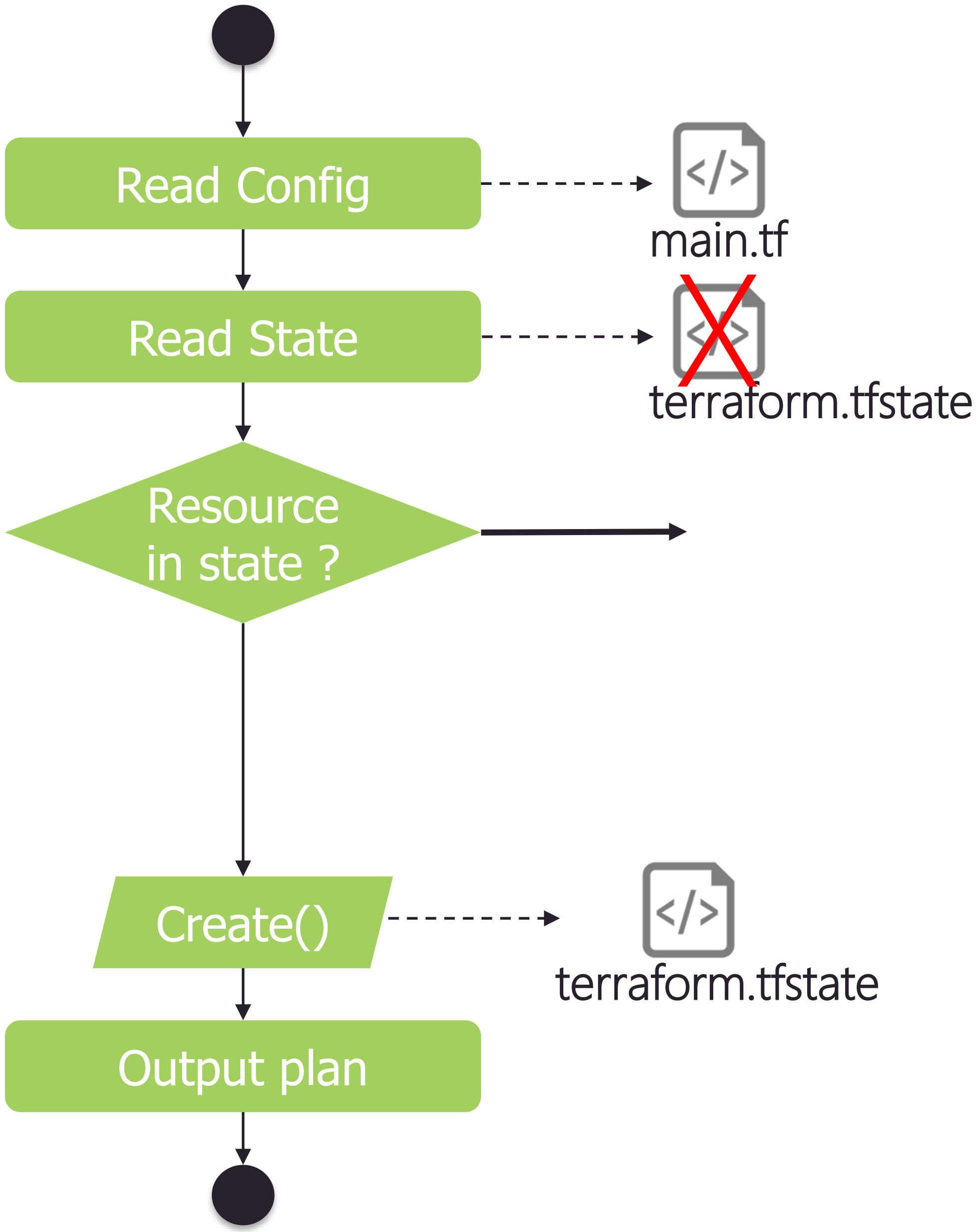
Creating the resource

```
terraform apply "plan.out"
```

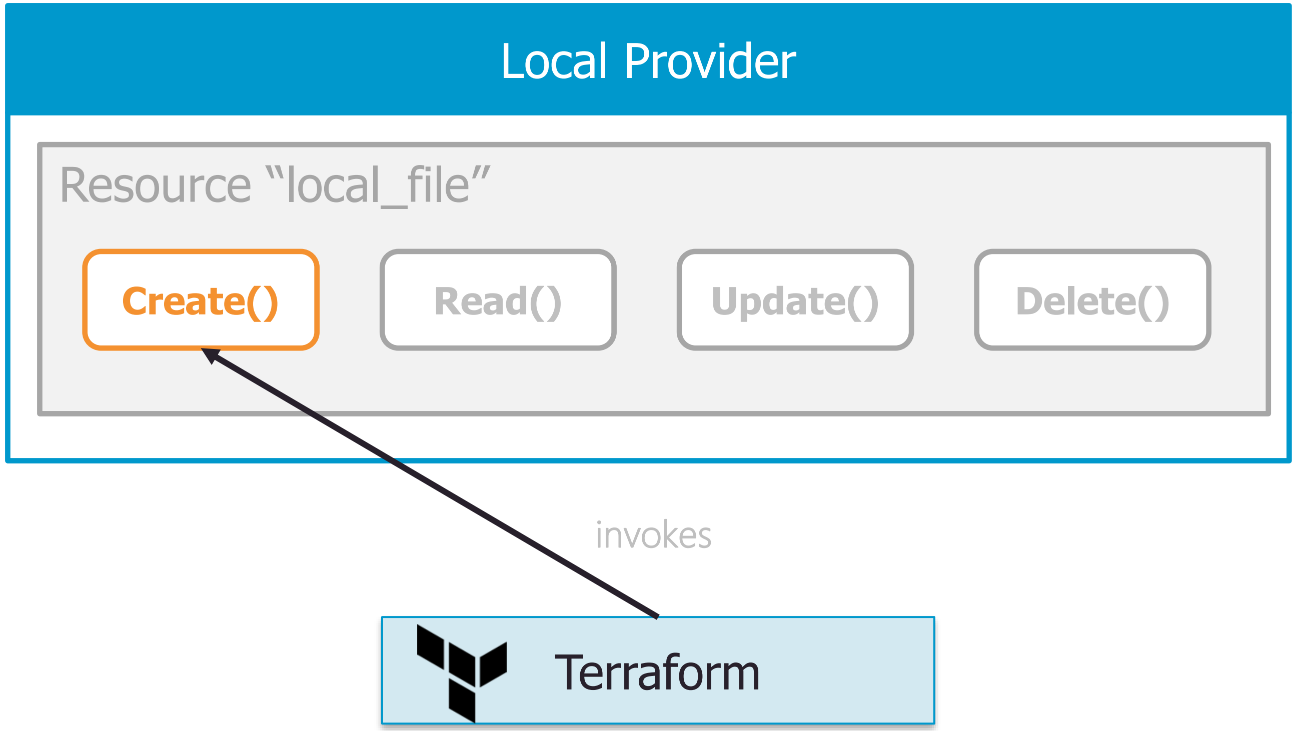
or

```
terraform plan -out plan.out && terraform apply "plan.out"
```

So far


Read Config
main.tf
Read State
terraform.tfstate
Resource
in state ?
Create()
terraform.tfstate
Output plan

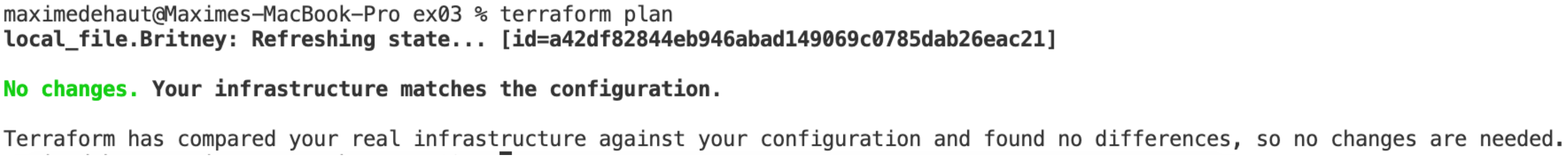
Behind the scene - Terraform apply



create.go

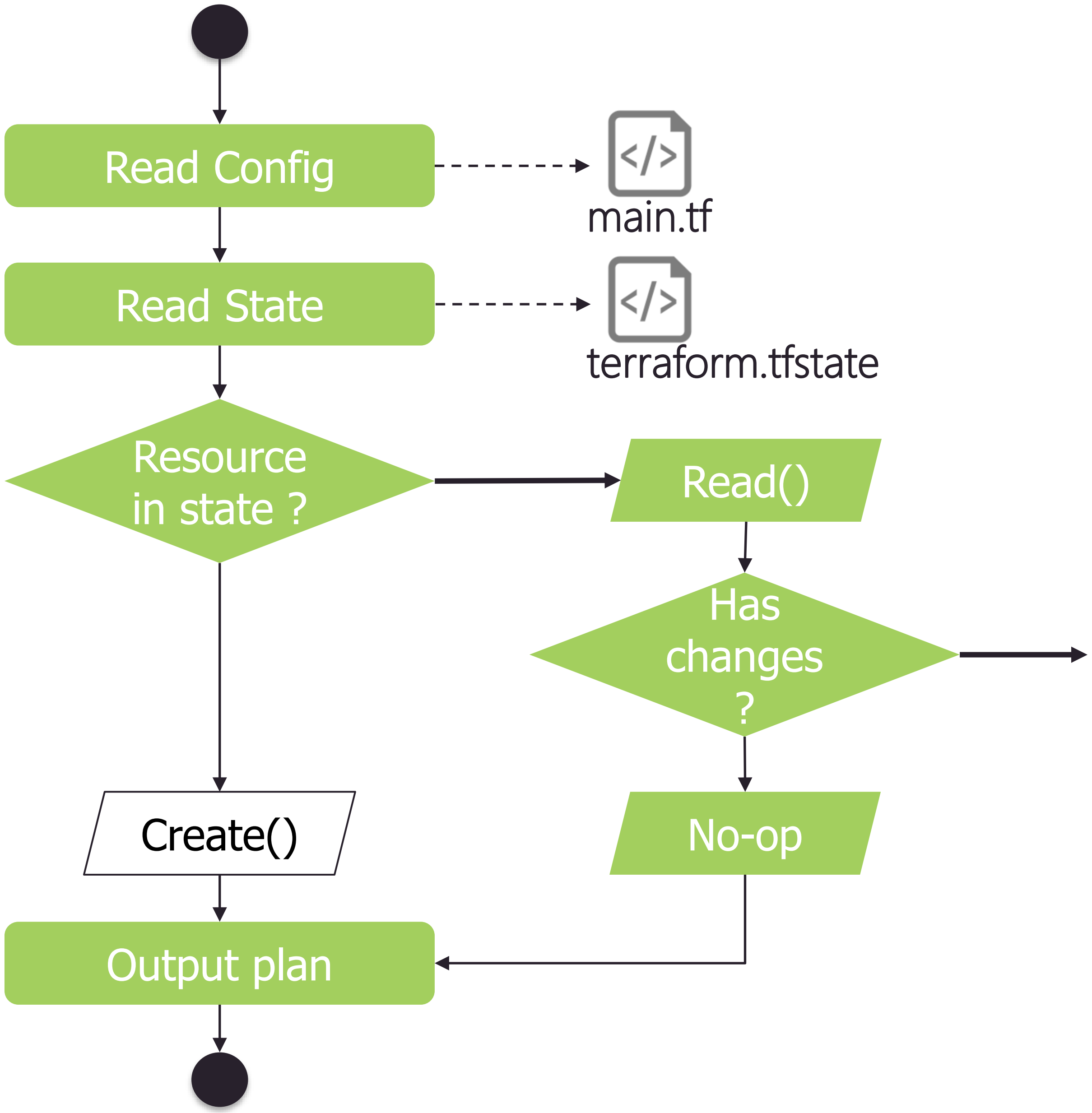
No-op() - redo

> terraform plan



```
maximedehaut@Maximes-MacBook-Pro ex03 % terraform plan
local_file.Britney: Refreshing state... [id=a42df82844eb946abad149069c0785dab26eac21]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.
```

So far


Read Config
main.tf
Read State
terraform.tfstate
Resource in state ?
Read()
Has changes ?
Create()
No-op
Output plan

Terraform states

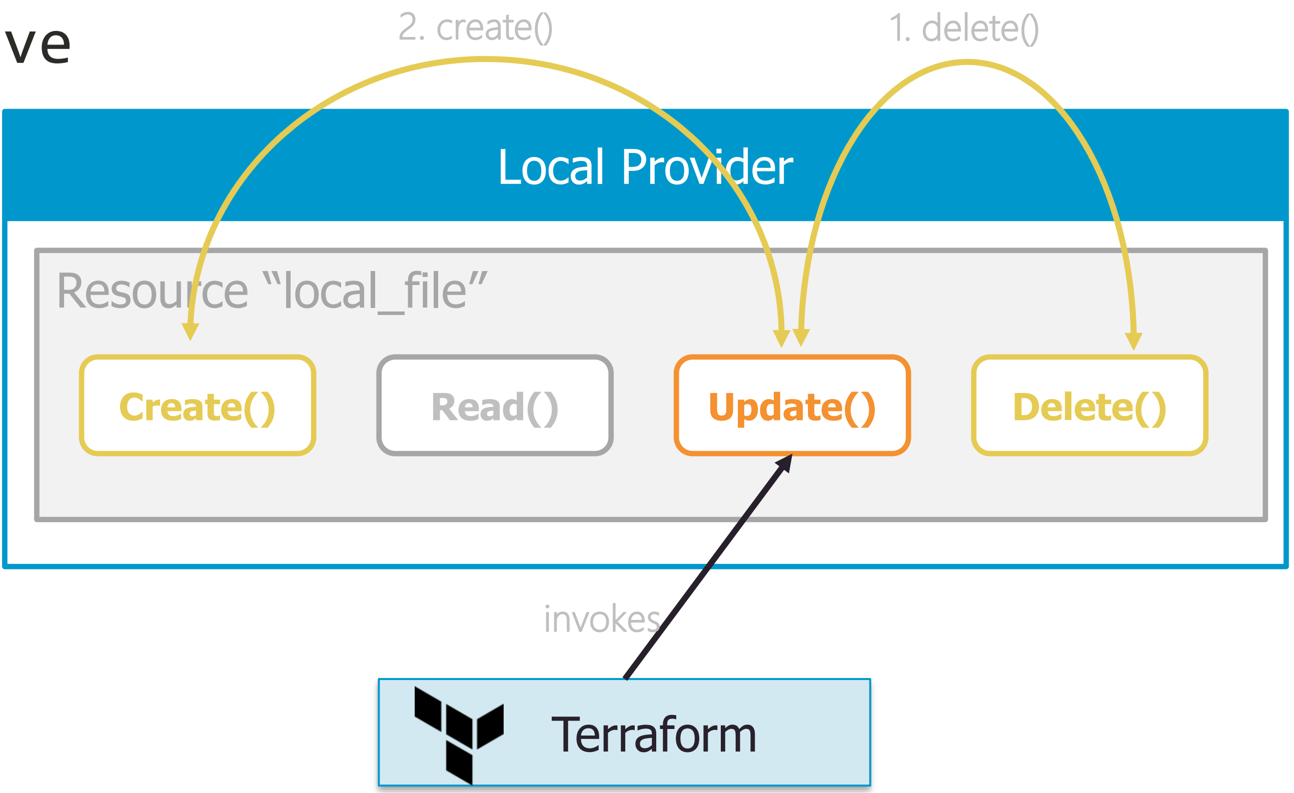
- Store stateful information about only three configuration blocks

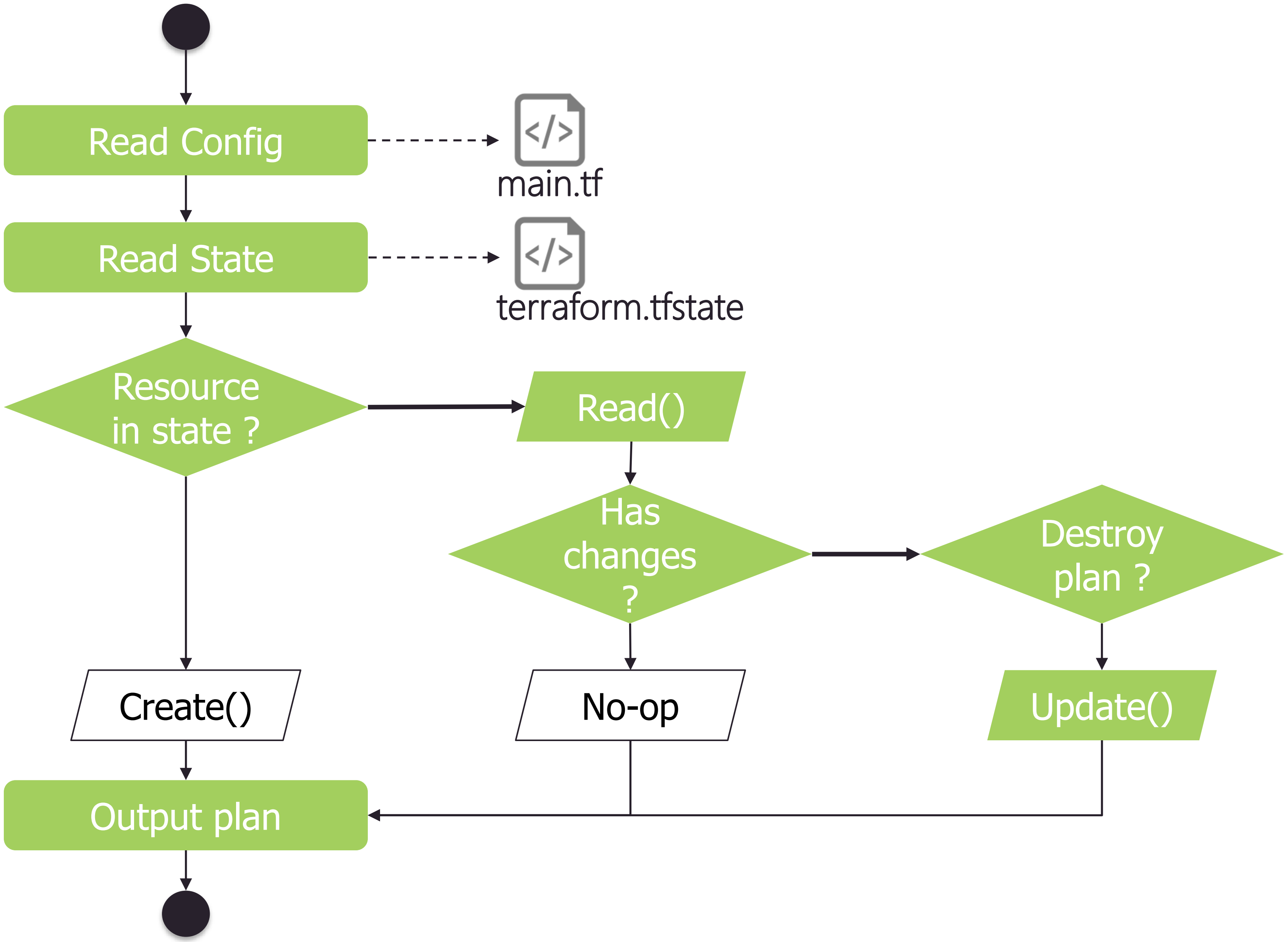
- `terraform state list`
- `terraform show`

- But what about
  - Shared storage
  - Locking
  - Isolation

Immutable vs mutable update

- Update main.tf code or pass a variable
- `terraform plan`
- `terraform apply -auto-approve`




Read Config
main.tf
Read State
terraform.tfstate
Resource in state ?
Read()
Has changes ?
Destroy plan ?
Create()
No-op
Update()
Output plan

Configuration drift

- Change content of text file

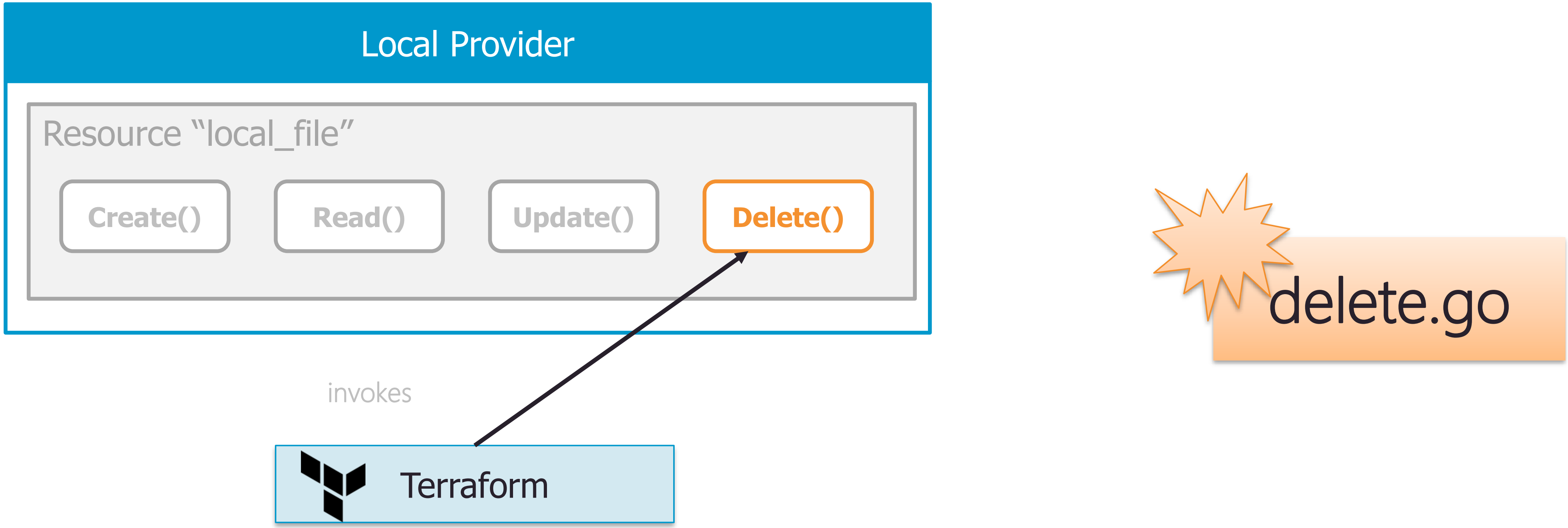
- `terraform plan`
- `terraform show`
- `terraform apply -refresh-only (= terraform refresh)`
  - `Existing resource deleted`
  - `New resource created`

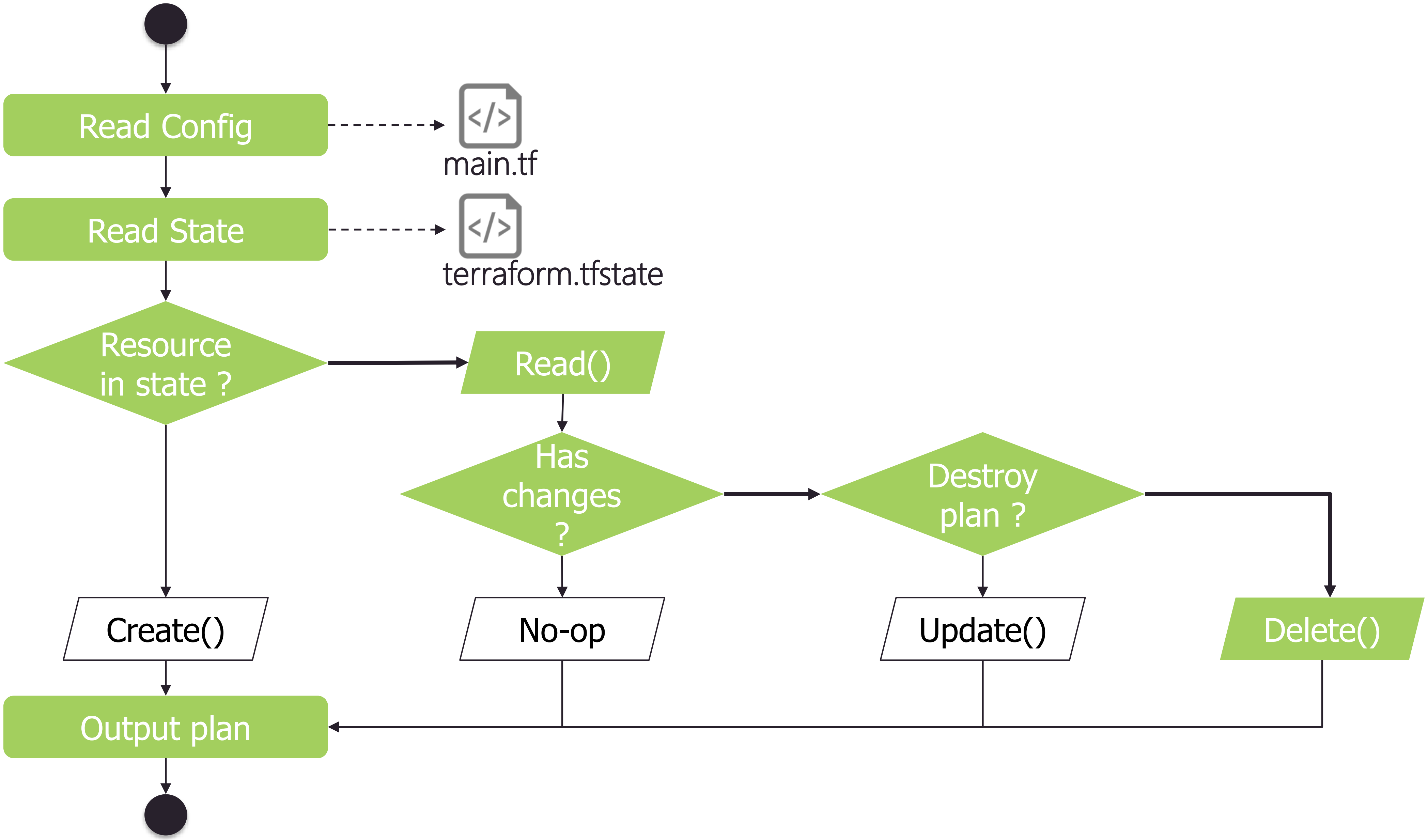
Show the state of each resource

reconcile the state

Deleting a resource

```
terraform destroy –refresh-only –auto-approve
```




Read Config
main.tf
Read State
terraform.tfstate
Resource in state ?
Read()
Has changes ?
Destroy plan ?
Create()
No-op
Update()
Delete()
Output plan

Note: Azure Authentication

- Authenticating to Azure using
  - The Azure CLI
  - Managed Service Identity
  - A Service Principal and
    - A client certificate
    - A client secret

Exercise 2.05 – Lifecycle in Azure

Check README file

> Note: for Windows users, graph command might generate some grey hair, therefore this step is optional

Exercise 2.06 – Configuration drift in Azure

Check README file

Getting Started with Terraform - Summary

- Terraform is a simple state management engine
- Resources are created in sequence dictated by the execution plan
- Terraform uses the state file during a plan

3. Functional Programming

Content

- Input variables, local values, and output values
- Making Terraform more expressive with functions and for expressions
- Incorporating two new providers: Random and Archive
- Templating with templatefile()
- Scaling resources with count

Functional programming ?

- Declarative programming paradigm
- Aggregation of modular functions

- Function’s attributes:
  - Pure functions
  - First-class and high-order functions
  - Immutability

Procedural s Functional

PROCEDURAL

```
const numList = [1, 2, 3, 4, 5]
let result = 0;
for(let i = 0; i < numList.length; i++) {
  if(numList[i] % 2=== 0) {
    result += (numList[i] * 10)
  }
}
```

FUNCTIONAL

```
const numList = [1, 2, 3, 4, 5]
const result = numList
               .filter(n=>n % 2=== 0)
               .map(a=>a * 10)
               .reduce((a, b) =>a + b)
```

TERRAFORM

```
locals {
  numList = [0, 1, 2, 3, 4, 5]
  result  = sum([for x in local.numList : 10 * x if x % 2 == 0])
}
```

Local value

- Assigns a name to an expression
- Allows multiple repetition

Element declaration

```
locals {
     …
}
```

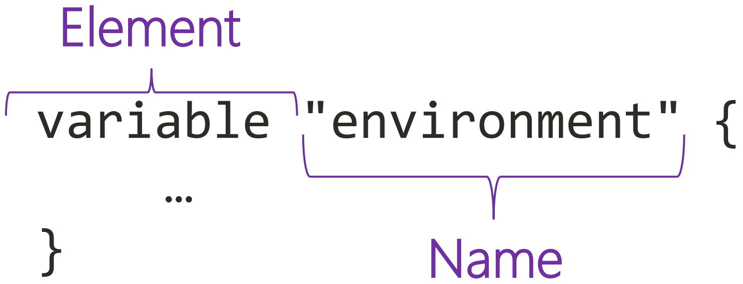
BUT

```
local.
```

When invoked

Input variable – Declaration

- The syntax of a variable block is



Element

```
variable "environment" {
      …
}
```

Name

- Variable values can be accessed via

When invoked: `var.environment`

Within a string: `${var.environment}`

- terraform apply -var variable_name="value"

Input variable – Arguments

- default
- description
- type
  - Primitive: string, integer, bool
  - Complex: list, map, set, object, tuple
- validation

```
variable "environment" {
    default = …
    description = …
    type = …
    validation {}
}
```

Input variable – Primitive types

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

```
variable "size" {
    type = number
    default = 123
}
```

```
variable "refresh_pwd" {
    type = bool
    default = false
}
```

Exercise 3.01 – Azure Resource Group Name

Check README file

Input variable – Collection types

```
variable "roles" {
     type = list(string)
     default = ["admin", "user"]
}
```

```
role = var.roles[0]
```

```
variable "plans" {
     type= map
     default = {
          "basic" = "1xCPU-1GB"
          "heavy" = "1xCPU-2GB"
     }
}
```

```
sizing = var.plans["basic"]
```

or

```
size = lookup(var.plans, "basic")
```

Input variable – Structural types

```
variable "user" {
    type = object({
        login = string                # a required attribute
        name  = optional(string)      # an optional attribute
    })
}
```

Validating Variables

- Validation block

```
variable "login" {
    type = string

    validation {
        condition = (var.login == var.login)
        error_message = "This is an error message."
    }
}
```

Validating Variables - Example

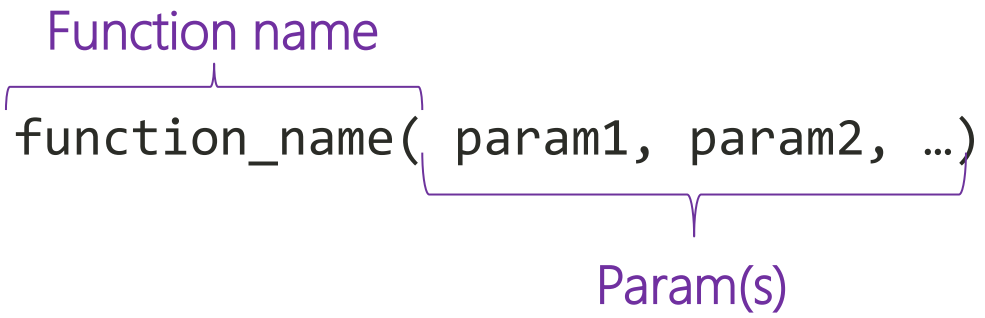
```
validation {
     condition = (length(var.login) >= 8)
     error_message = ”Login does not match expected length."
}

validation {
     condition = (length(var.accepted_envs) <= 3)
     error_message = ”Nbr of environment is too high."
}

validation {
     condition = length(var.rgName) >= 6 && substr(var.rgName, 0, 3) == "rg-"
     error_message = ”Must start with a 'rg-' and contains at least 6 chars."
}

...
```

Exercise 3.02 – Validation rule

Check README file

Functions

- Terraform functions are expressions that transform inputs into outputs.
- Restricted to built-in functions.
- Terraform extension is done through customised provider



Function name

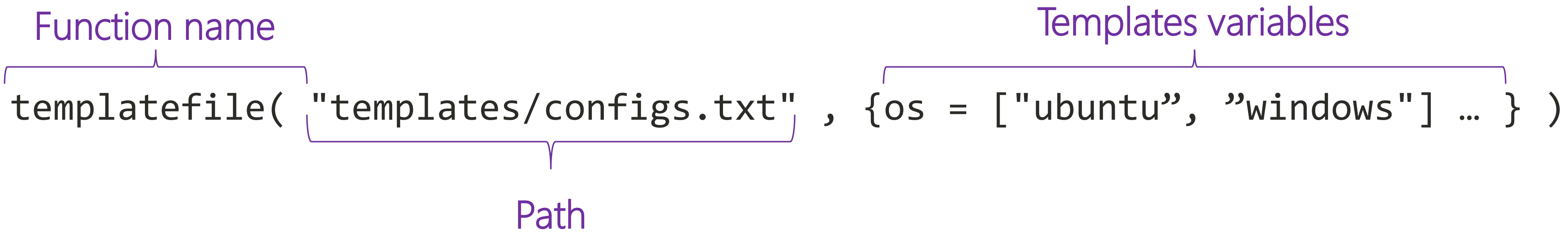
```
function_name( param1, param2, …)
```

Param(s)

```
length(…)
contains(…, …)
```

Function – templatefile()

- templatefile() used to replace placeholder values in a template file



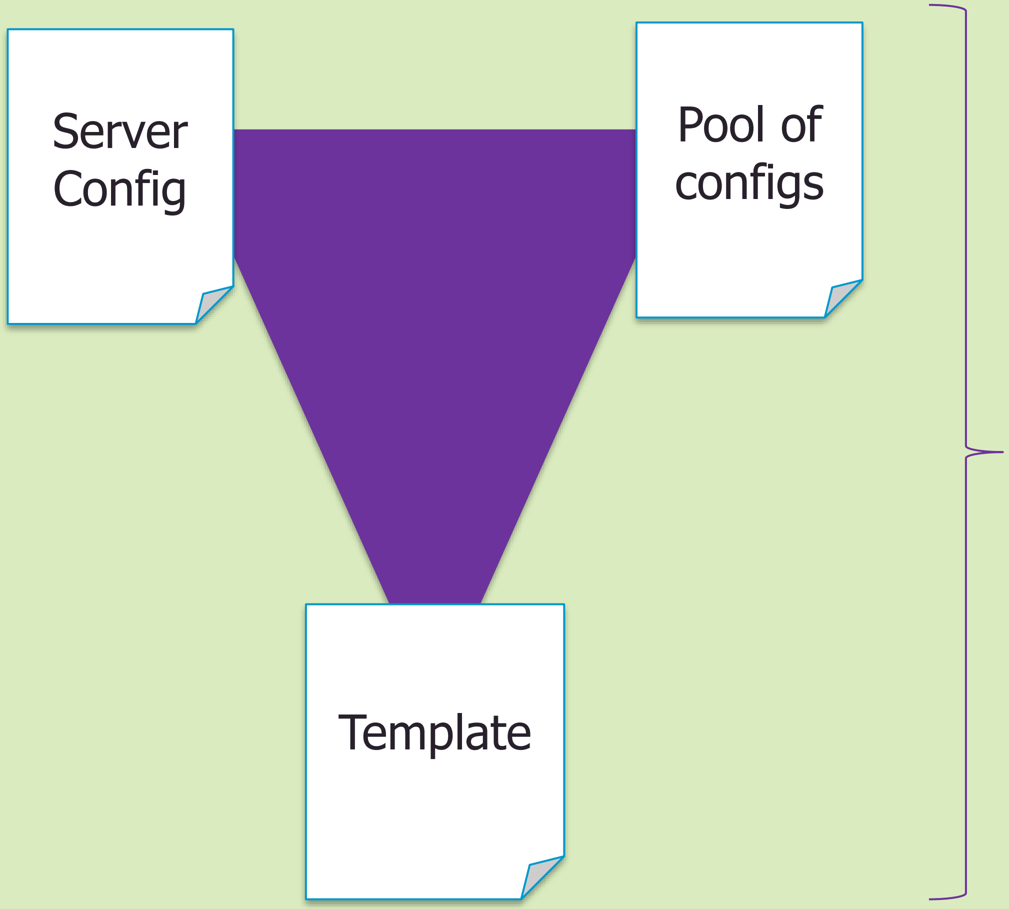
Function name

Templates variables

```
templatefile( "templates/configs.txt" , {os = ["ubuntu”, ”windows"] … } )
```

Path

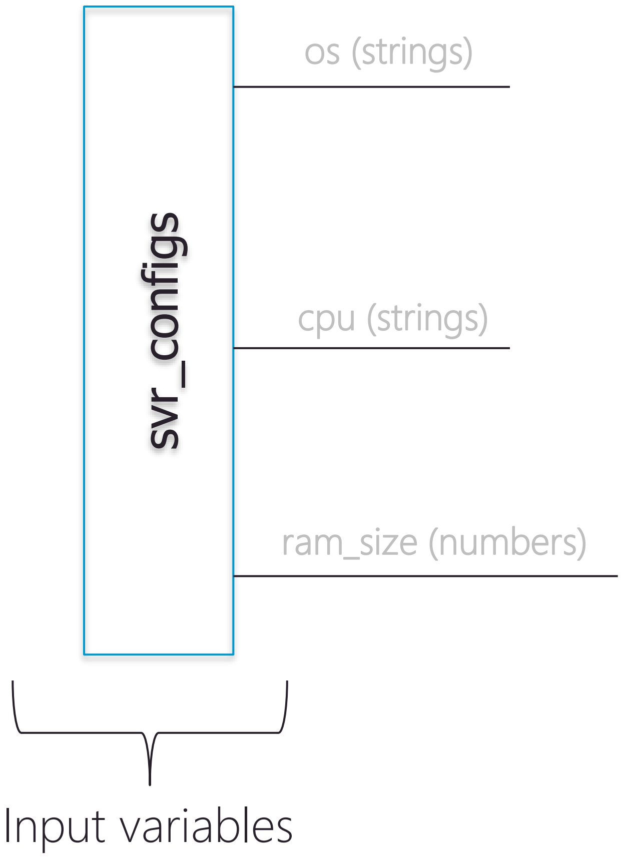
Example 3.03 – Description



Rendering in CLI

Server Config(s)

TYPE COERCION

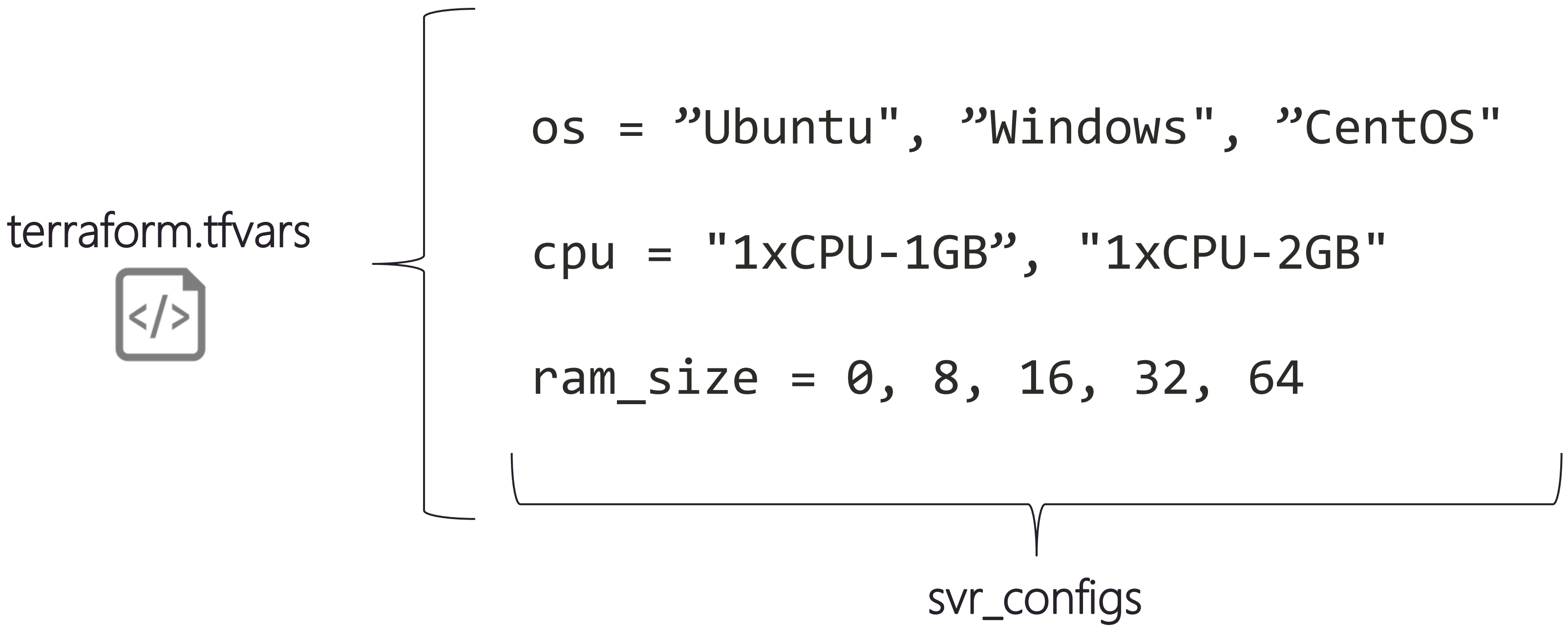


```
terraform {
  required_version = ">= 0.15”
}

variable "svr_configs" {
    description = "A list of svr config"
    type = object ({
        os = list(string),
        cpu = list(string),
        ram_size = list(number),
    })
}
```

Exercise – Add a validation rule to check that there is at least 3 ram_sizes

Assigning Values with a Variable Definition File

- File ending in either .tfvars or .tfvars.json



terraform.tfvars

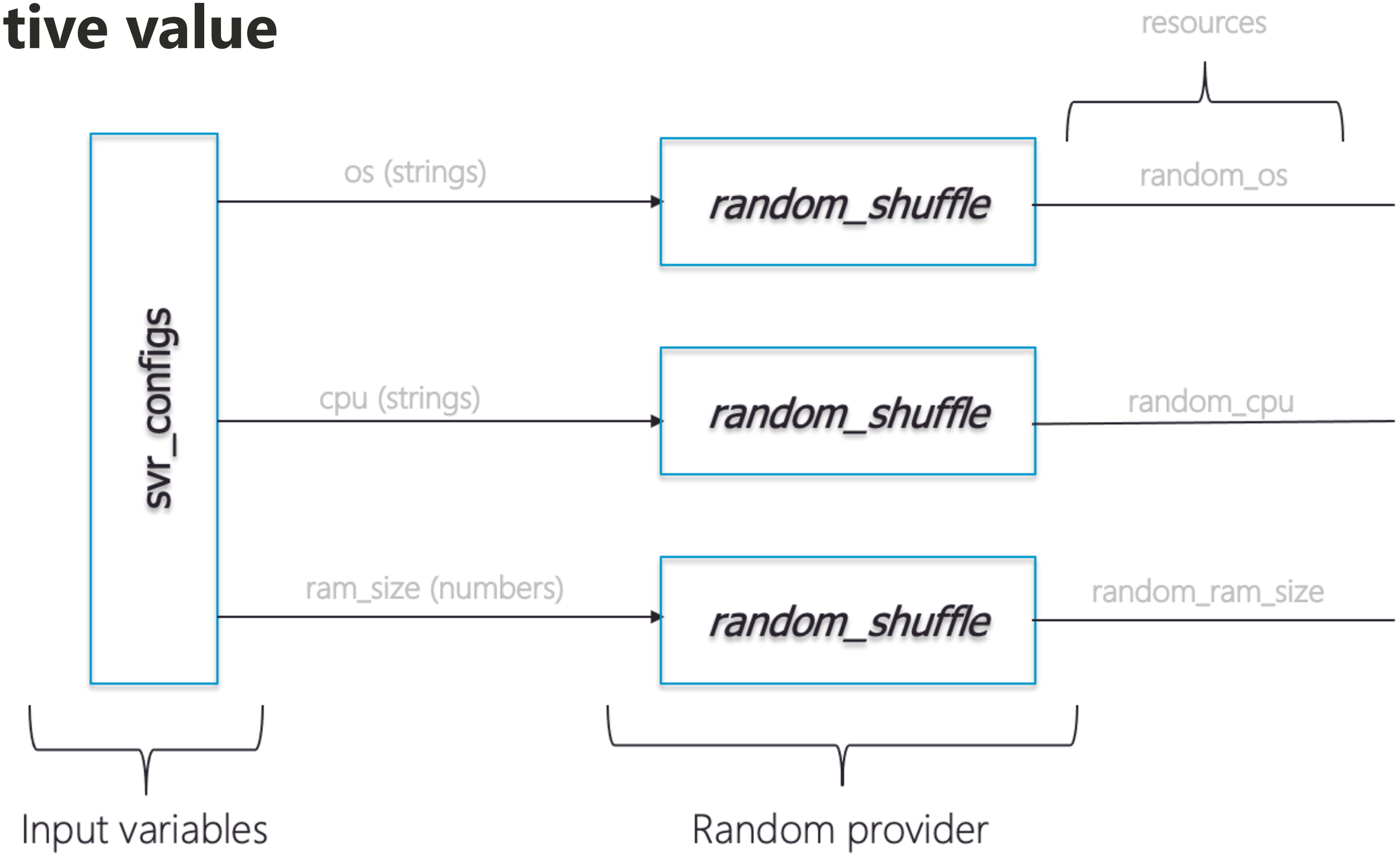
```
os = ”Ubuntu", ”Windows", ”CentOS"

cpu = "1xCPU-1GB”, "1xCPU-2GB"

ram_size = 0, 8, 16, 32, 64
```

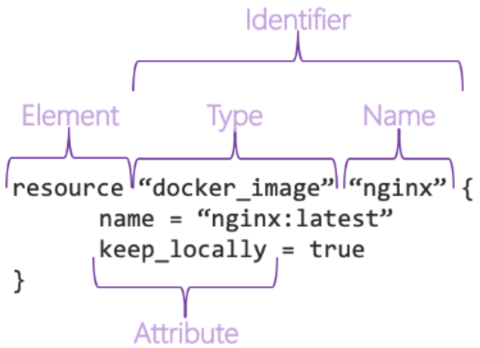
svr_configs

- **Exercise – Create a pool of configs file**

Server Config

Exercise – Create the resources and set the 'input' attribute with the respective value





```
resource "docker_image" "nginx" {
     name = "nginx:latest"
     keep_locally = true
}
```

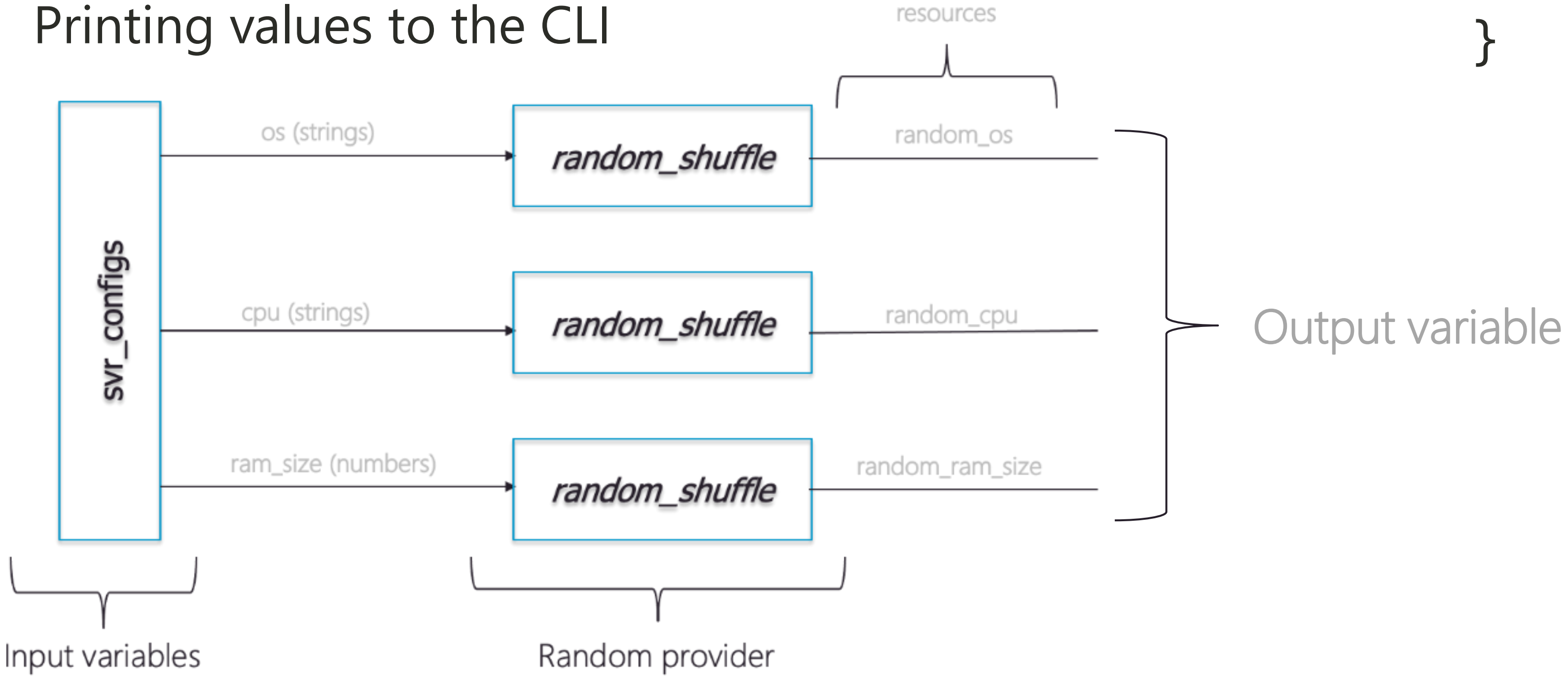
Output variable

- Output values are for doing one of two things:
  - Passing values between modules
  - Printing values to the CLI

Element

```
output “environment” {
        …
}
```

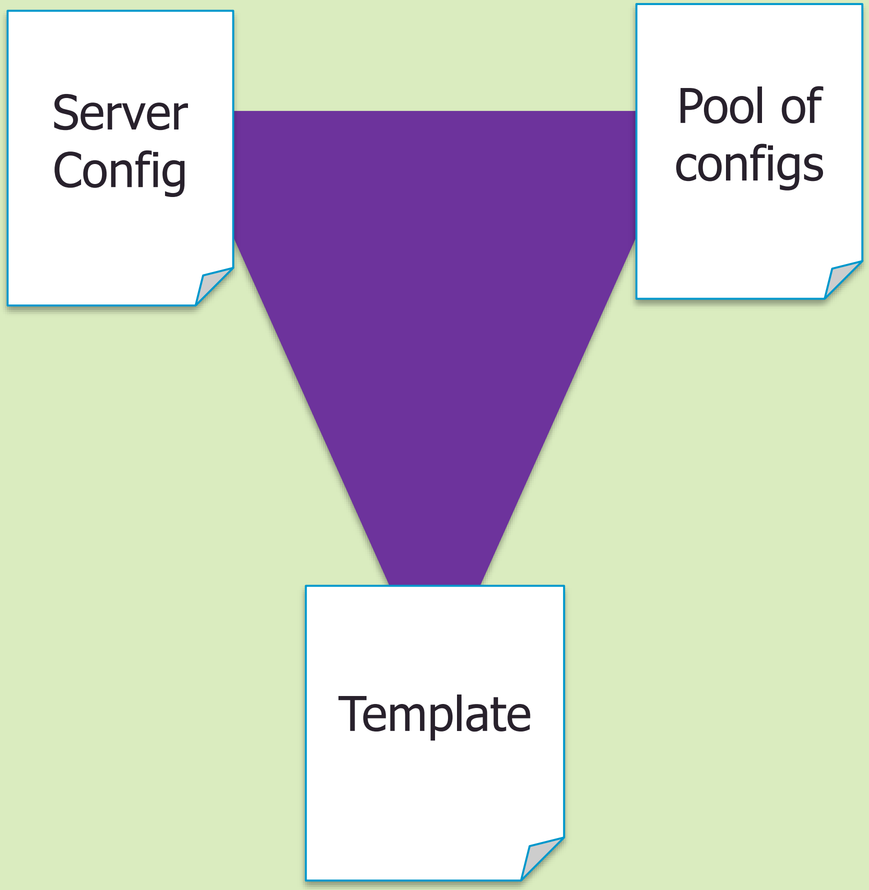
Name



- **Exercise – Create the output variable**

So far

```
terraform {
   required_version = ">= 0.15"
}

variable "svr_configs" {
   description = "A list of server config"
   type = object ({
      …
   })
   validation {
      …
   }
}

resource "random_shuffle" "random_os"{
   input = …
}

resource "random_shuffle" "random_cpu"{
   input = …
}

resource "random_shuffle" "random_ram_size"{
   input = …
}

output "out_cfg" {
}
```



```
svr_configs = {
   os = [ "ubuntu", "windows", "centos" ]
   cpu = [ "1xCPU-1GB", "1xCPU-2GB" ]
   ram_size = [0, 8, 16, 32, 64]
}
```

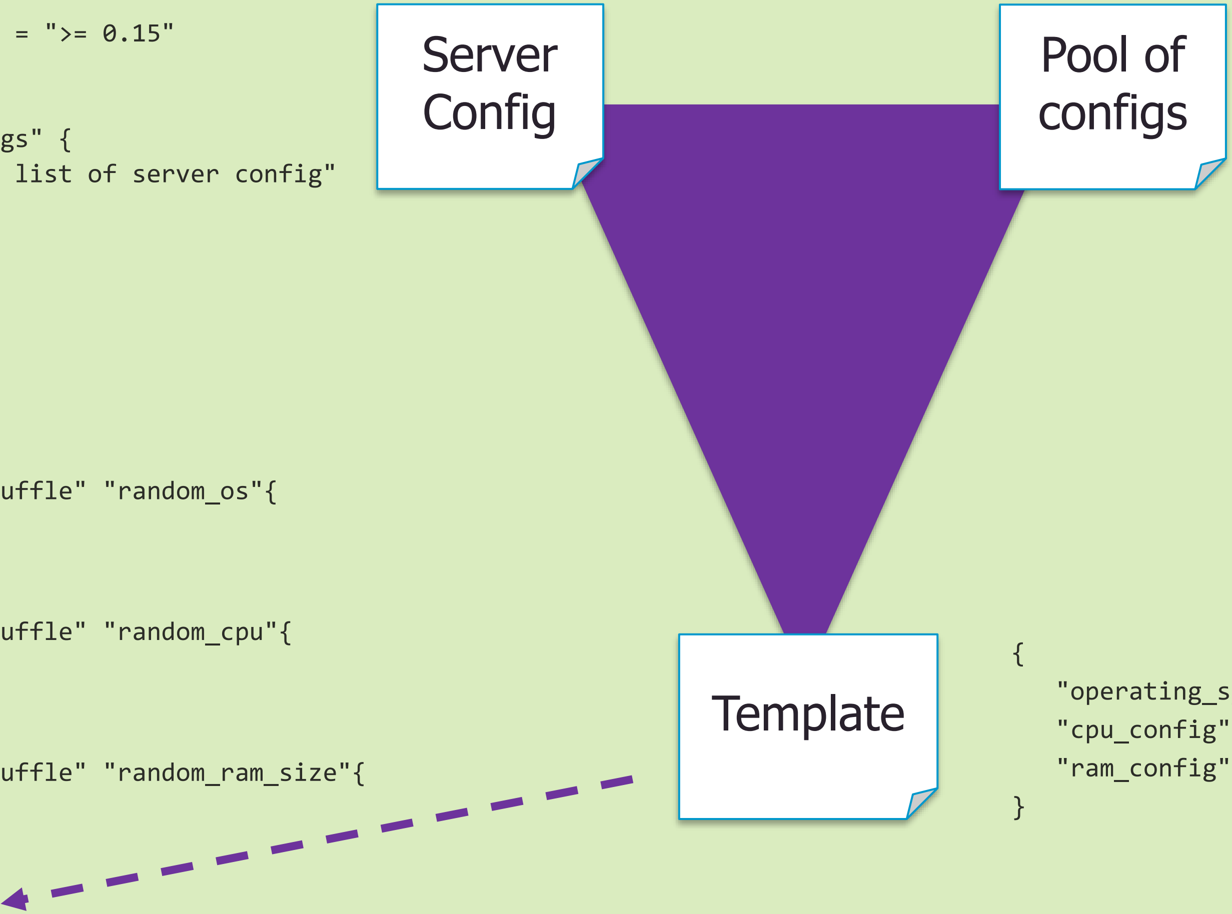
Template file

- Terraform syntax based

- **Exercise – Create a new directory called templates**
- **Exercise – In this directory, create a ”typical_svr.json” file representing a typical server: os, cpu, ram_size**

So far

```
terraform {
   required_version = ">= 0.15"
}

variable "svr_configs" {
   description = "A list of server config"
   type = object ({
      …
   })
   validation {
      …
   }
}

resource "random_shuffle" "random_os"{
   input = …
}

resource "random_shuffle" "random_cpu"{
   input = …
}

resource "random_shuffle" "random_ram_size"{
   input = …
}

output "out_cfg" {
   value = …
}
```



```
svr_configs = {
   os = [ "ubuntu", "windows", "centos" ]
   cpu = [ "1xCPU-1GB", "1xCPU-2GB" ]
   ram_size = [0, 8, 16, 32, 64]
}
```

```
{
   "operating_system": "${os[0]}",
   "cpu_config": "${cpu[0]}",
   "ram_config": ${ram_size[0]}
}
```

Random provider

Exercise – Declare the required provider within the terraform element

```
required_providers {
     random= {
          source= "hashicorp/random"
          version = "~> 3.0"
     }
}
```

Let’s roll

```
terraform init && terraform apply -auto-approve
```

What happens if you apply twice ? Why ?

Example 3.04 – Generating multiple configs

- Comment output element
- Create 2 additional templates files

Local file – Prepare

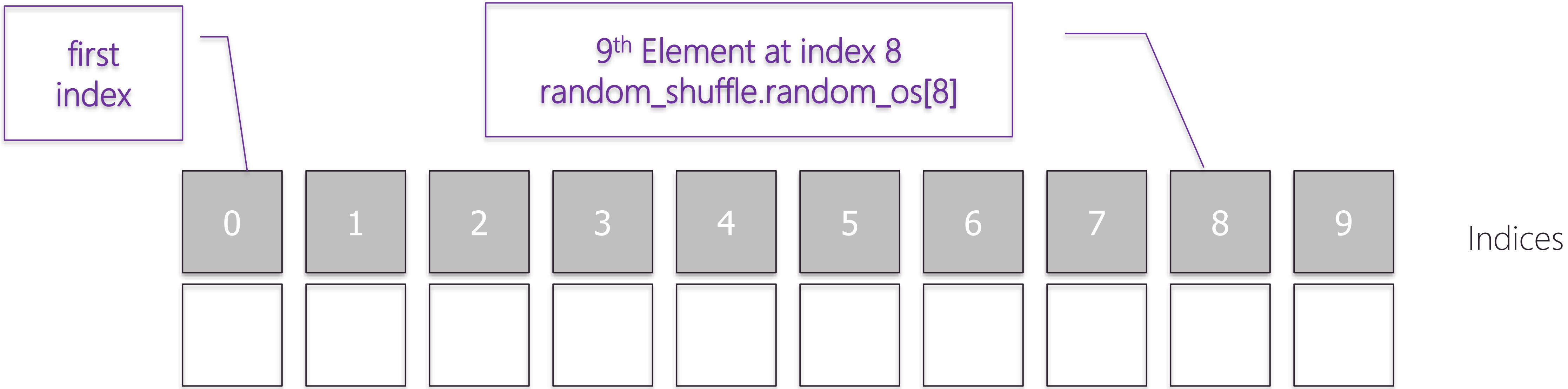
- Read all the template files from the template folder into a set

```
fileset(path.module, "templates/*.json"))
```

- Convert set into a list and declare it in locals

```
locals {
    templates = tolist(fileset(path.module, "templates/*.json"))
}
```

Counter parameter



first index

9th Element at index 8
random_shuffle.random_os[8]

0	1	2	3	4	5	6	7	8	9	Indices

- **Exercise – Add a new variable named var.num_files having type number and a default value of 10**
- **Exercise – Reference this variable to dynamically set the count meta argument on each of the shuffle_resources & the file name**

Archive the files

Exercise – Declare the required provider within the terraform element

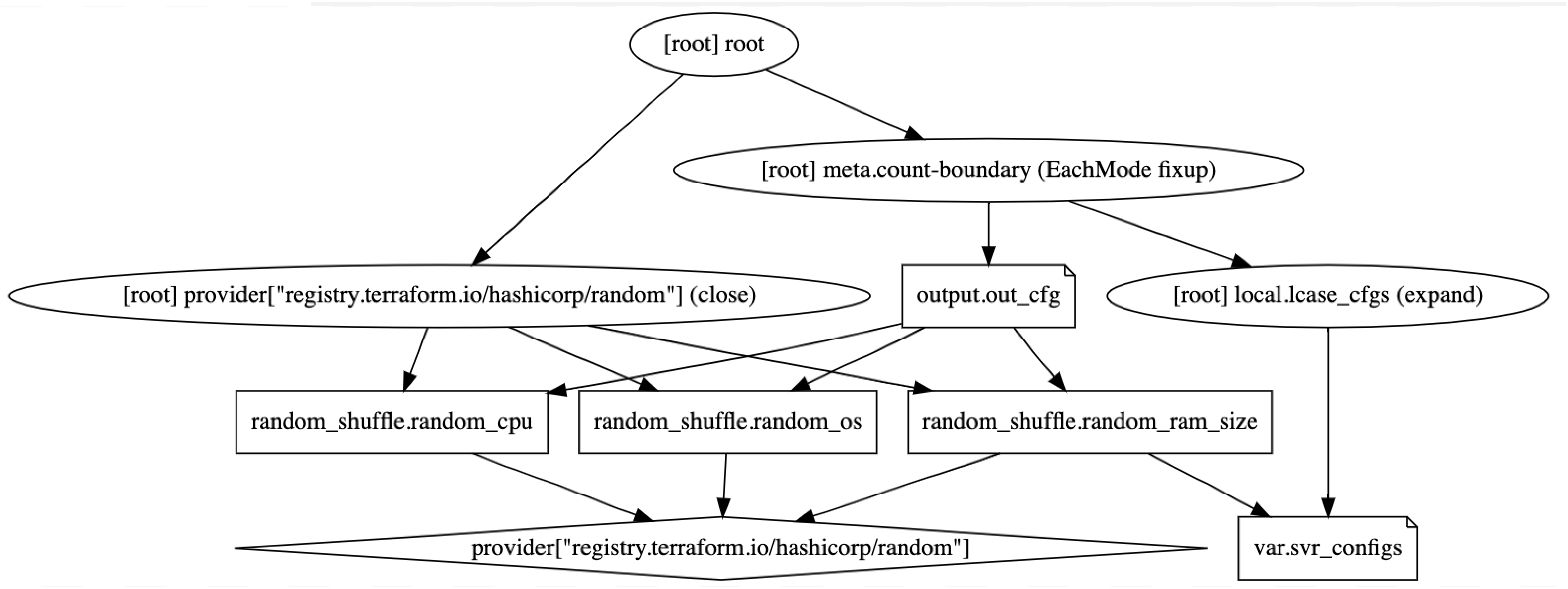
```
required_providers {
     archive= {
          source= "hashicorp/archive"
          version = "~> 2.0"
     }
}
```

Archive the files – cont’d

Exercise – Declare a data element representing the archive file

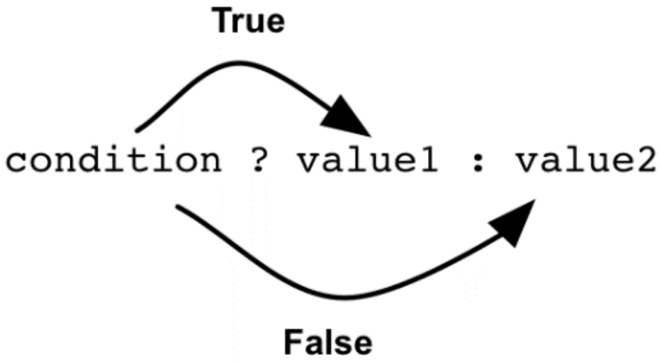
```
data "archive_file" "zip_cfg" {
     type = "zip"
     source_dir = "${path.module}/configs"
     output_path = "${path.cwd}/config.zip"
     depends_on = [local_file.out_cfg]
}
```

Note: Implicit dependencies


[root] root
[root] meta.count-boundary (EachMode fixup)
[root] provider["registry.terraform.io/hashicorp/random"] (close)
output.out_cfg
[root] local.lcase_cfgs (expand)
random_shuffle.random_cpu
random_shuffle.random_os
random_shuffle.random_ram_size
provider["registry.terraform.io/hashicorp/random"]
var.svr_configs

Note: Conditional expressions

- Conditional expressions hurt readability a lot, so avoid using them if you can



True

`condition ? value1 : value2`

False

```
locals {
    v = length(var.svr_configs["os"])>=1 ? var.svr_configs["os"] : [][0]
}
```

Exercise 3.05 – Conditional expression

Check README file

So far

- Input variables, local values, output values
- For expressions
- Randomness must be constrained
- Zip at runtime – explicit dependency
- templatefile()
- Count meta argument

Terraform expressions

Name	Description	Example
Conditional Expressions	Use the value of a boolean expression to select one of two values	condition ? true_value : false_value
Function Calls	Transform and combine values	<FUNCTION NAME>(<ARG 1>, <ARG2>)
For Expression	Transform one complex type to another	[for s in var.list : upper(s)]
Split Expressions	Shorthand for some common use cases that could otherwise be handled by for expressions	var.list[*].id equivalent for expression: [for s in var.list : s.id]

Dynamic Blocks	Construct repeatable nested blocks within resources	dynamic "ingress" {
		  for_each = var.service_ports
		  content {
		    from_port = ingress.value
		    to_port = ingress.value
		    protocol = "tcp"
		  }
		}

String Template Interpolation	Embed expressions in a string literal	"Hello, ${var.name}!"

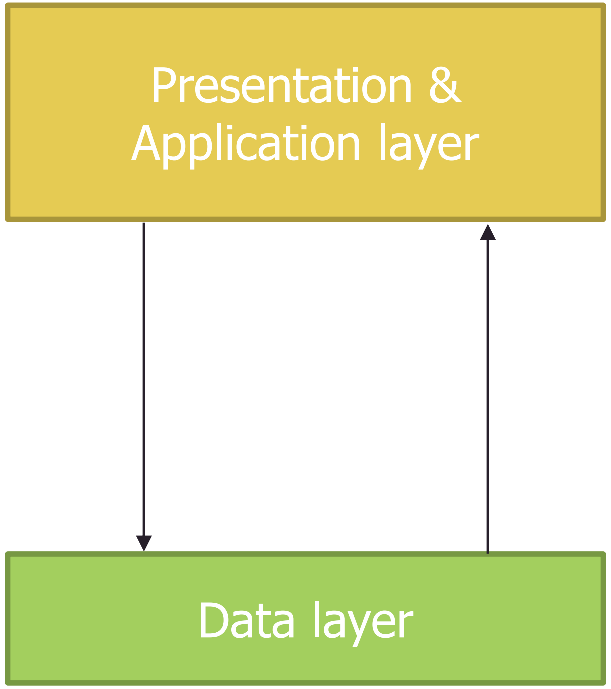
String Template Directives	Use conditional results and iteration over a collection within a string literal	%{ for ip in var.list.*.ip }
		server ${ip}
		%{ endfor }

4. Application Deployment

Content

- Deploying a multi-tiered web application in Azure with Terraform
- Setting project variables in a variable's definition files
- Organizing code with nested modules
- Leveraging modules from the public module registry
- Passing data between modules using input variables and output values

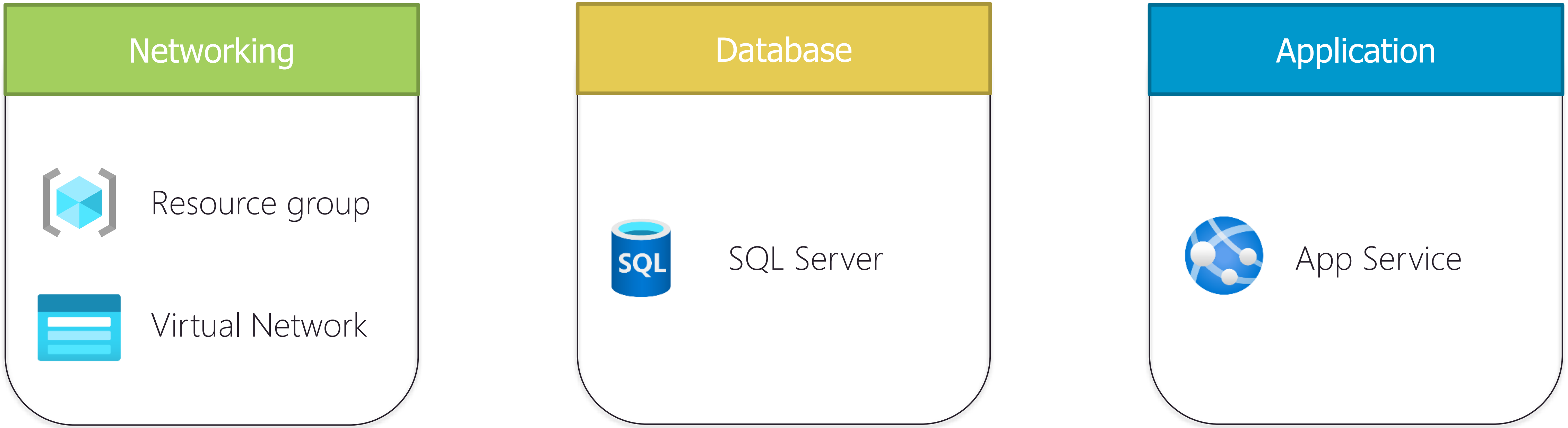
Multi-Tiered Web Application in Azure



HTML/CSS/JS
REST API

Database

Architecture



Networking

Resource group

Virtual Network

Database

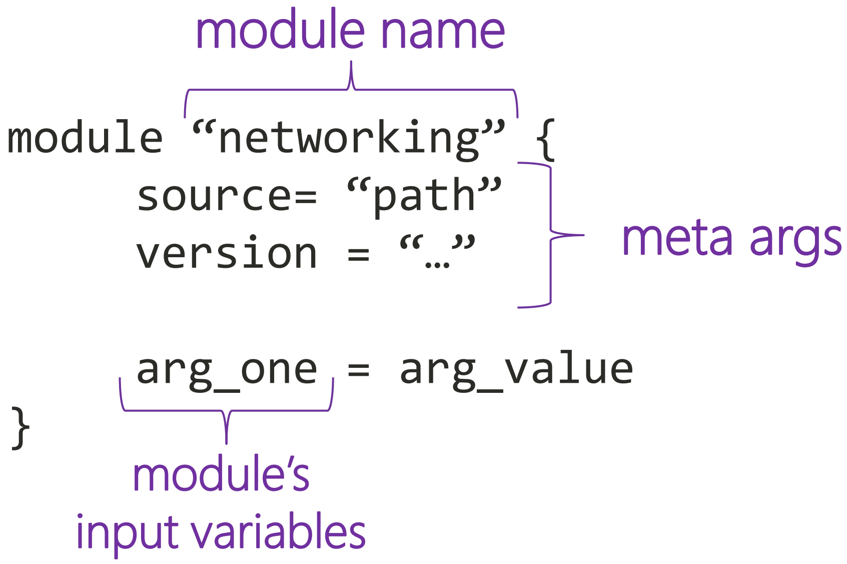
SQL Server

Application

App Service

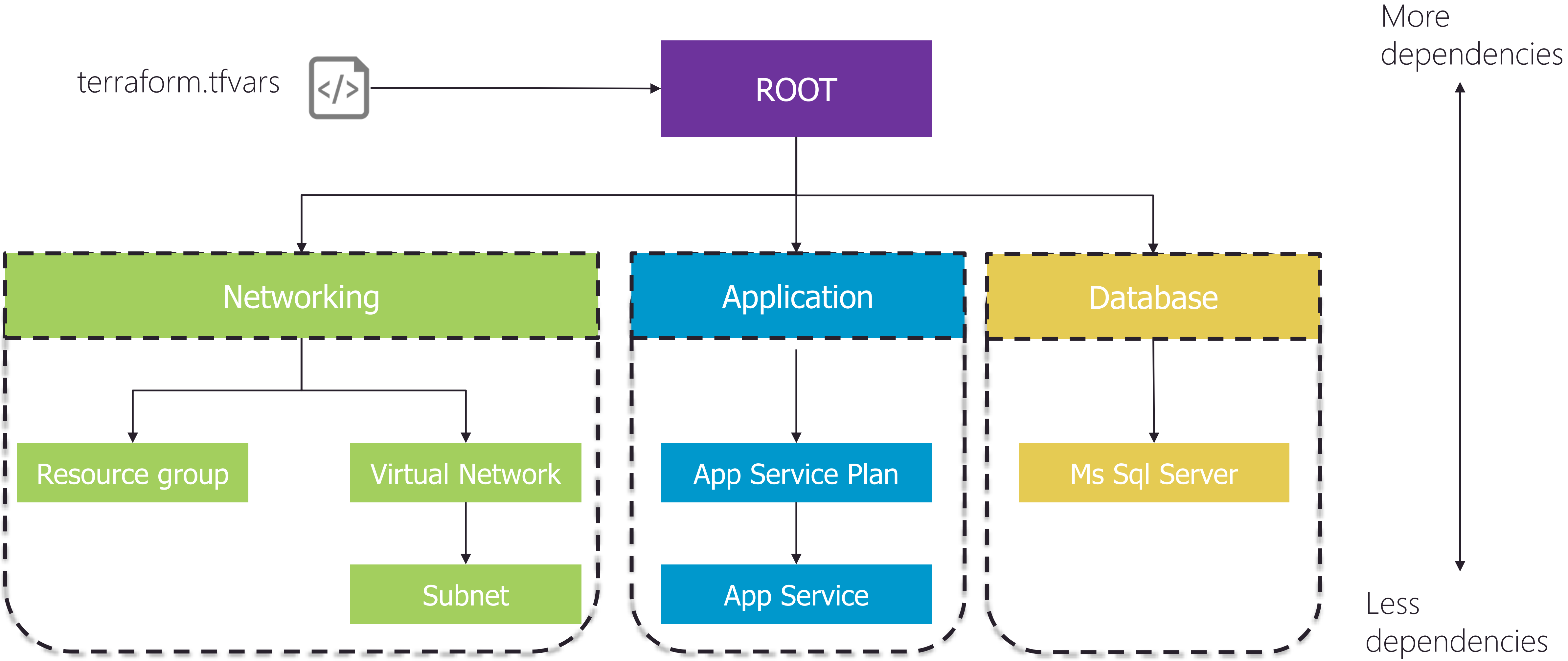
Terraform Module

- Definition
  - Self-contained packages of terraform code
  - Consume inputs, produces outputs
  - Allow/Use for code reuse and software abstraction
- Syntax

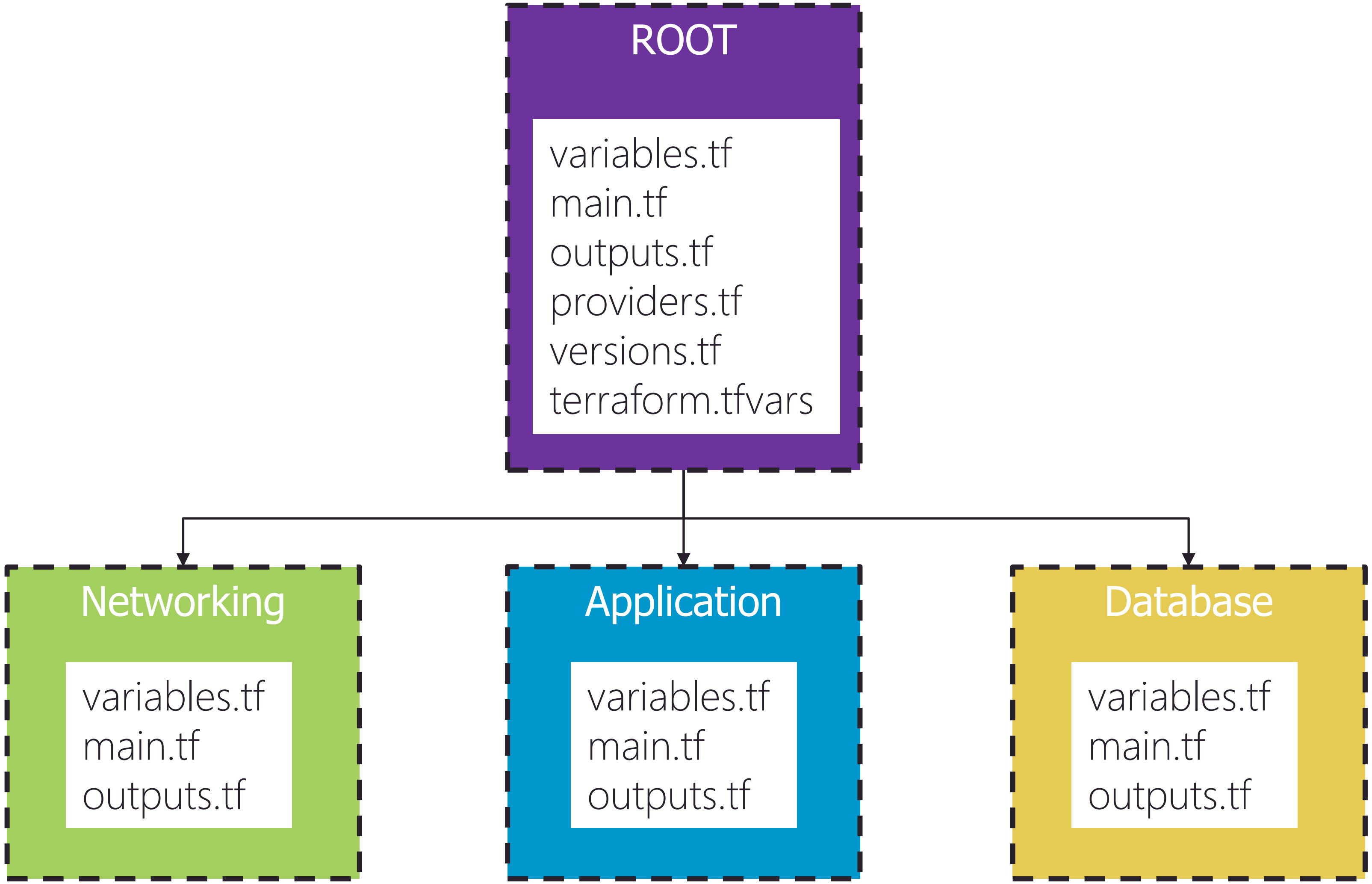


```
module “networking” {
     source= “path”
     version = “…”

     arg_one = arg_value
}
```

Approach by Terraform



Recommended structure



ROOT

variables.tf
main.tf
outputs.tf
providers.tf
versions.tf
terraform.tfvars

Networking

variables.tf
main.tf
outputs.tf

Application

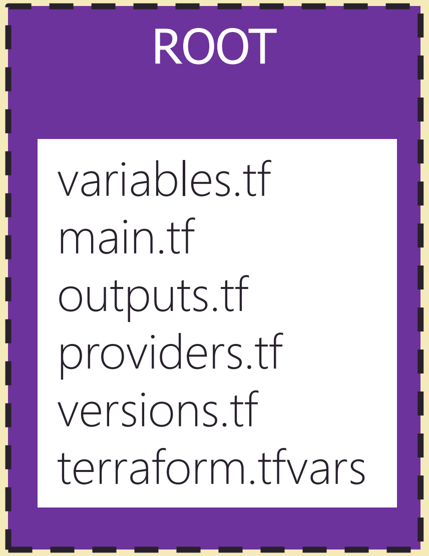
variables.tf
main.tf
outputs.tf

Database

variables.tf
main.tf
outputs.tf

Exercise 4.01 – Define root module

Check README file

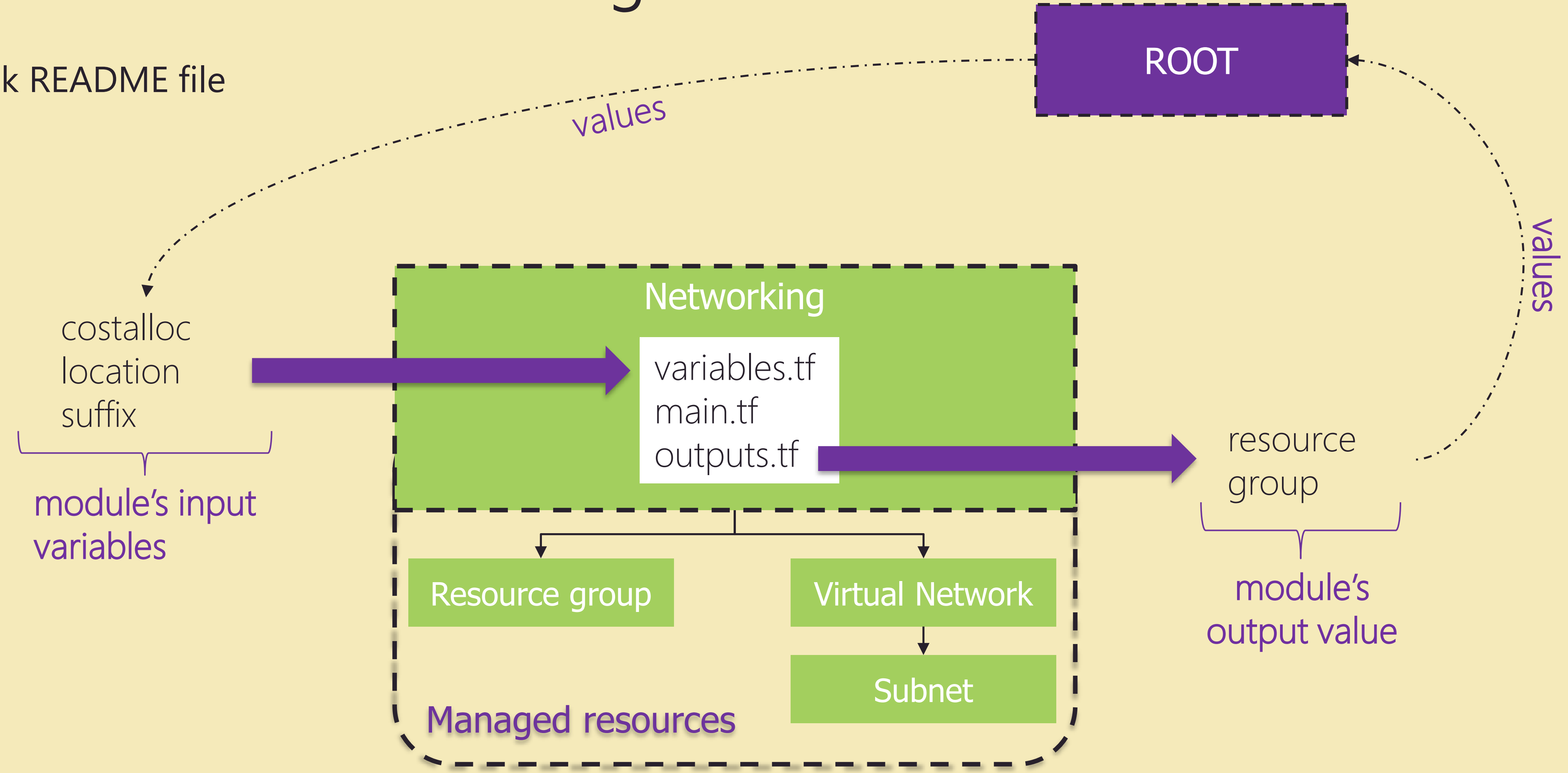


ROOT

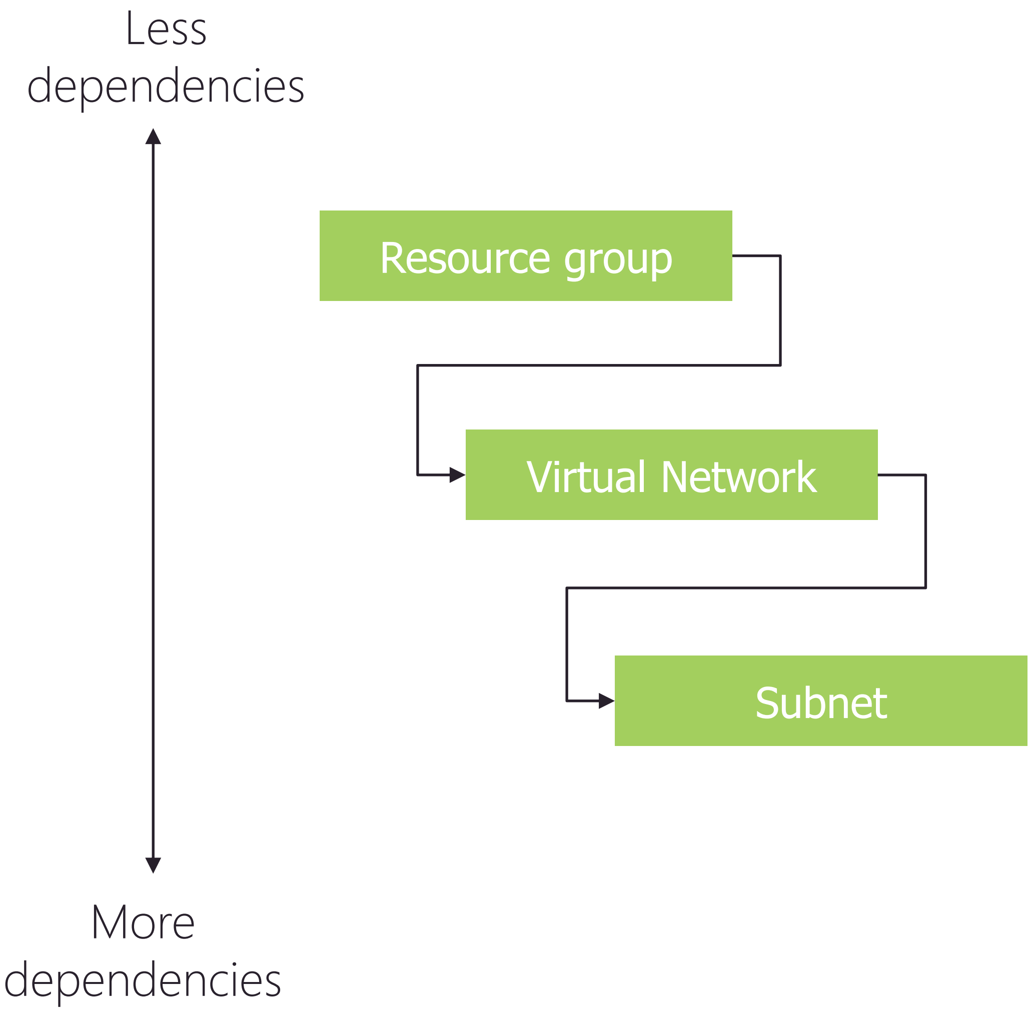
variables.tf
main.tf
outputs.tf
providers.tf
versions.tf
terraform.tfvars

Exercise 4.02 – Networking module

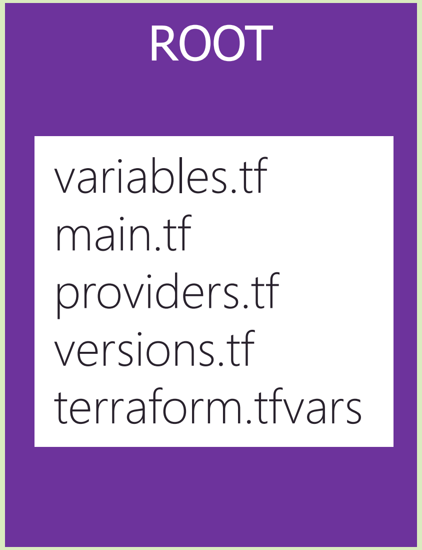
Check README file



Networking module – Structure



Example 4.03 – Software Componentization



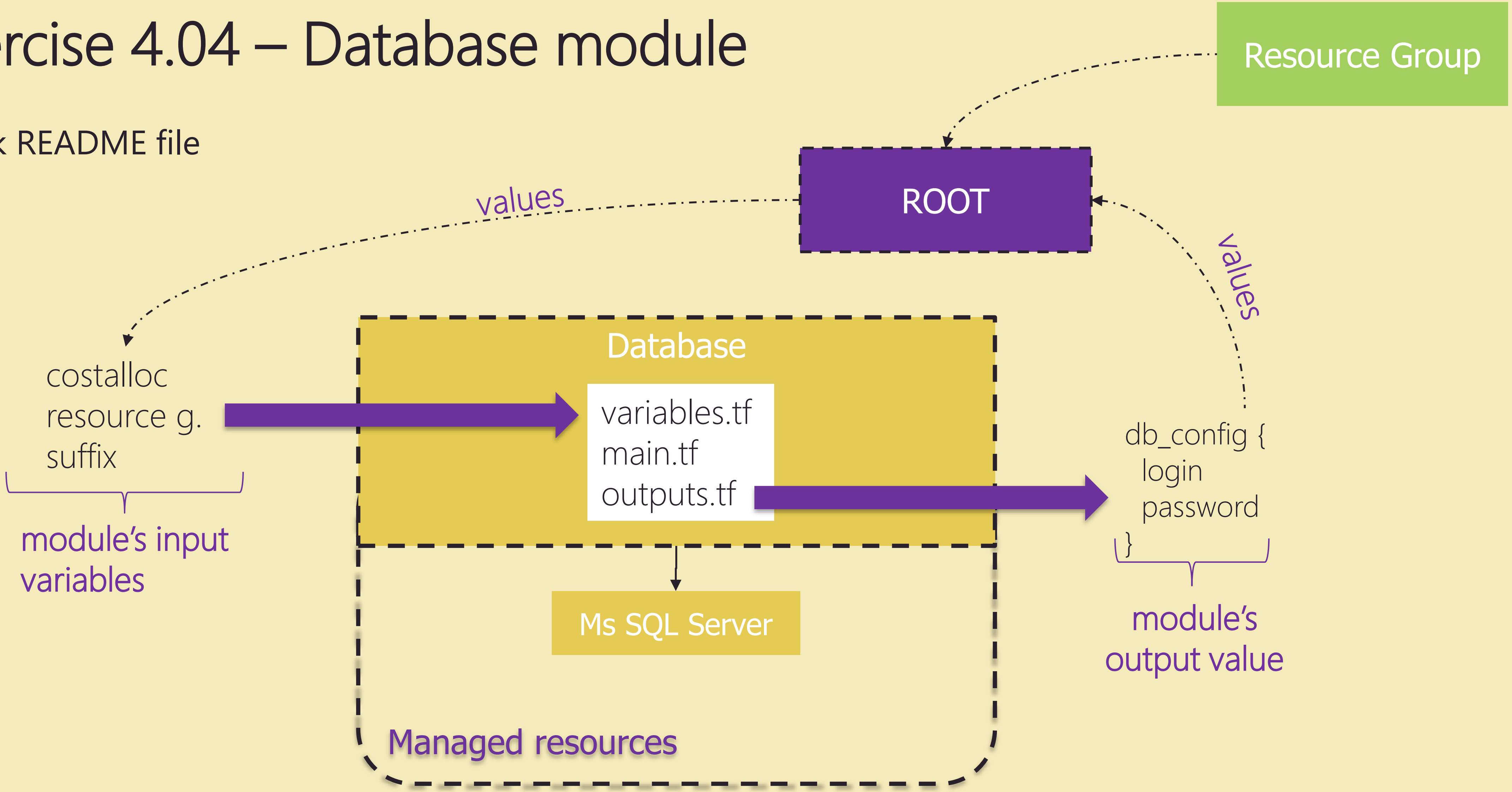
ROOT

variables.tf
main.tf
providers.tf
versions.tf
terraform.tfvars

- Existing module
  - Azure/vnet/azurerm | Terraform Registry
- Source code (see in Section 5):
  - Azure/terraform-azurerm-vnet: Terraform module to create/provision Azure vnet (github.com)

Exercise 4.04 – Database module

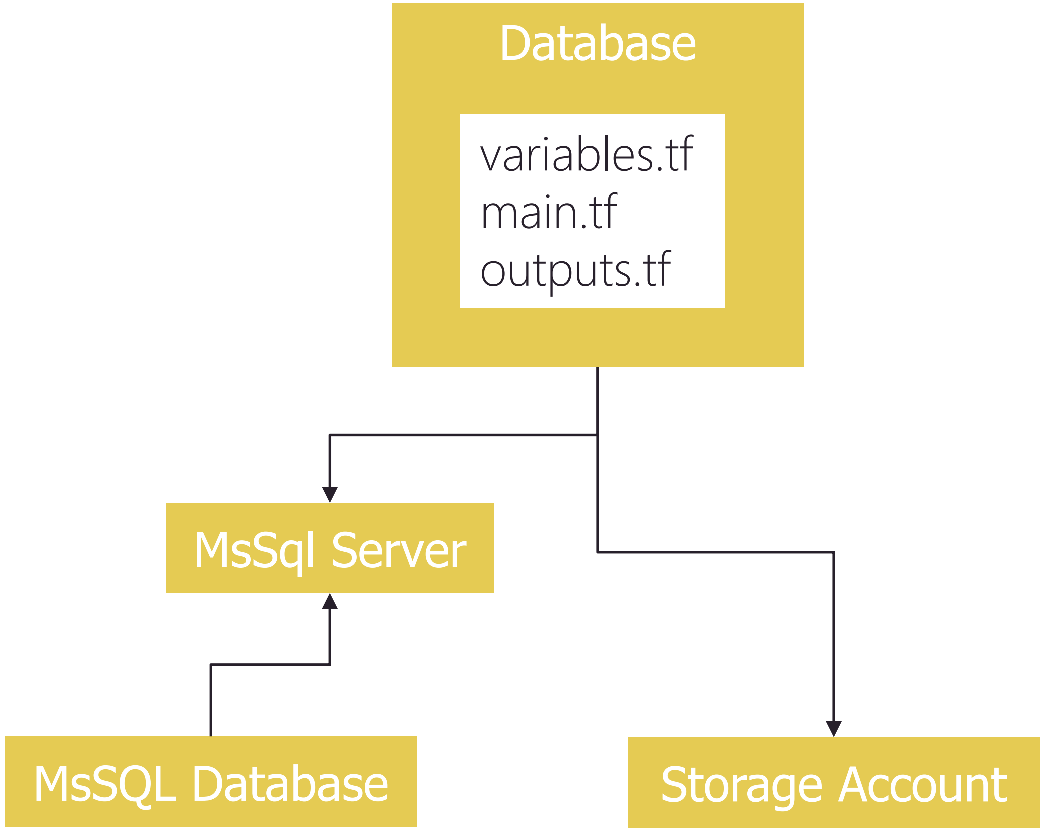
Check README file



Database module – Ex 4.04 (bonus)

- Add an MsSQL database
- Add a storage account

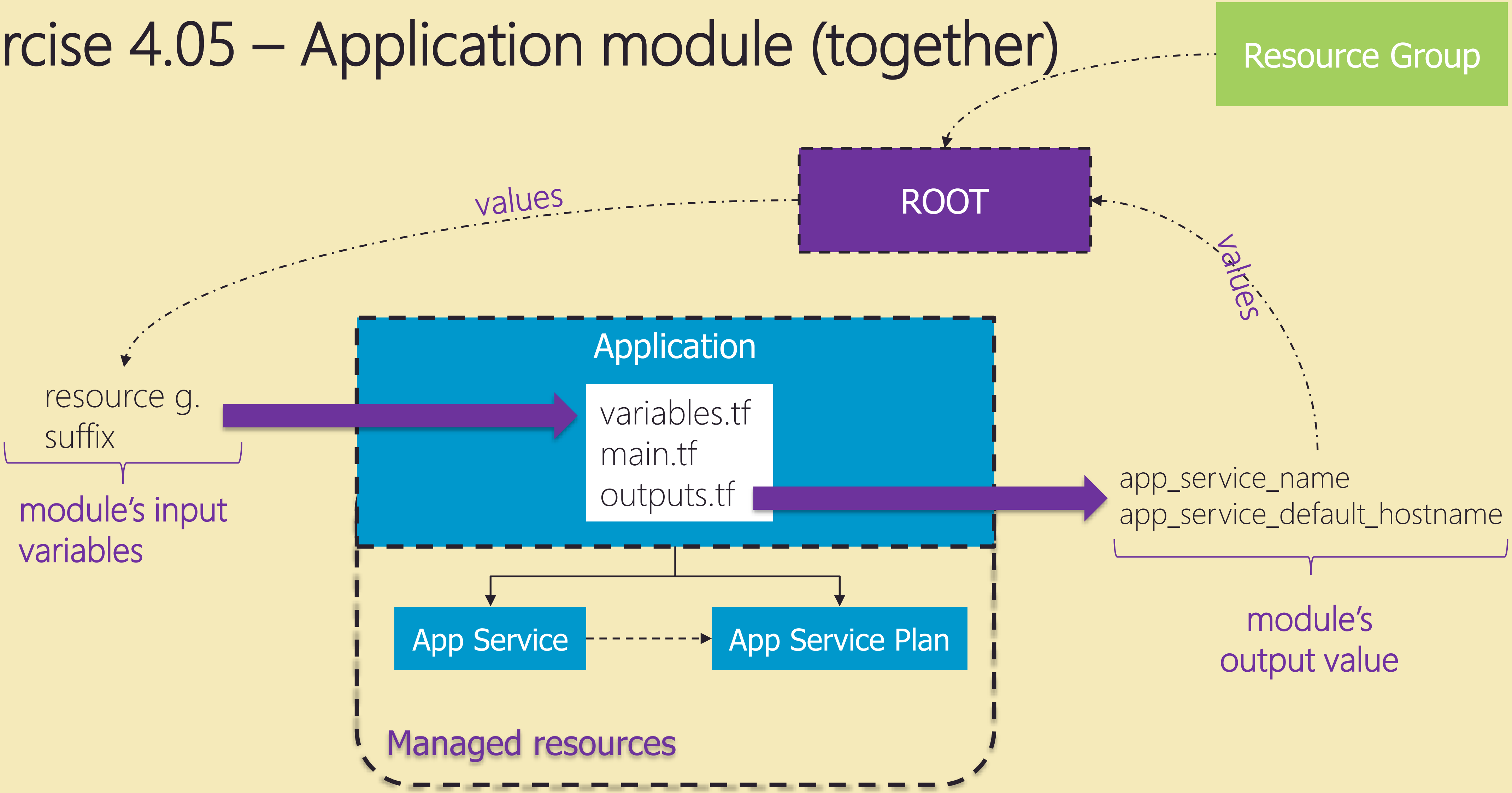
- *Check in Terraform Registry*



Display output values

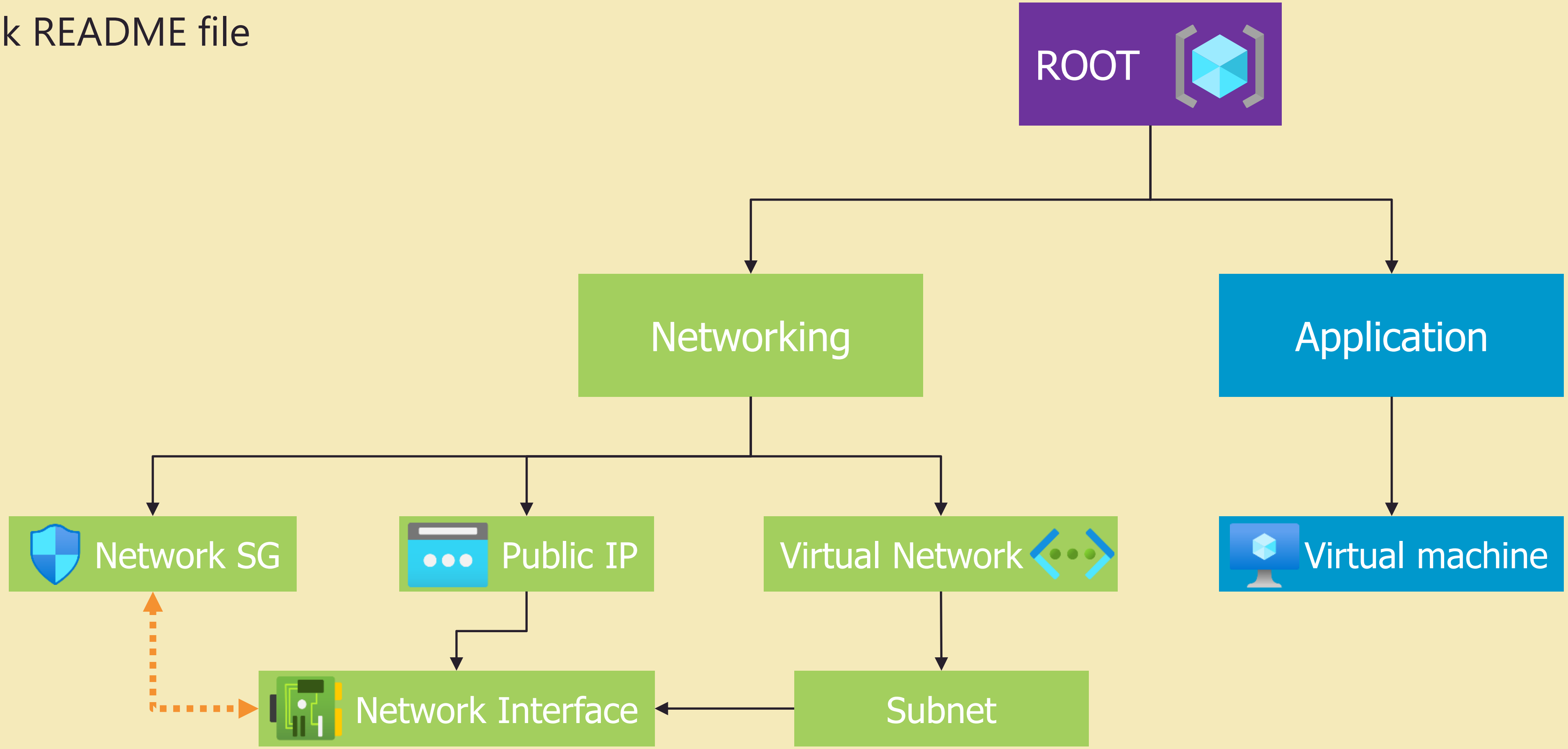
```
terraform output «output value»
```

Exercise 4.05 – Application module (together)



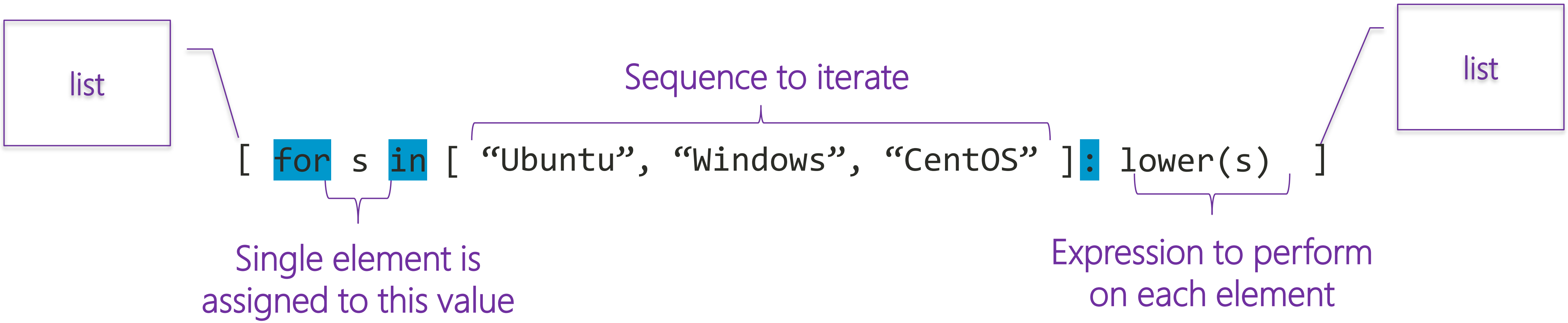
Exercise 4.06 – Typical infrastructure

Check README file



"For" Expressions

- Are anonymous functions that can transform one complex type into another
- Simple expressions can be composed to construct higher-order functions
- They use lambda like syntax and are comparable to lambda expressions and streams



list

Sequence to iterate

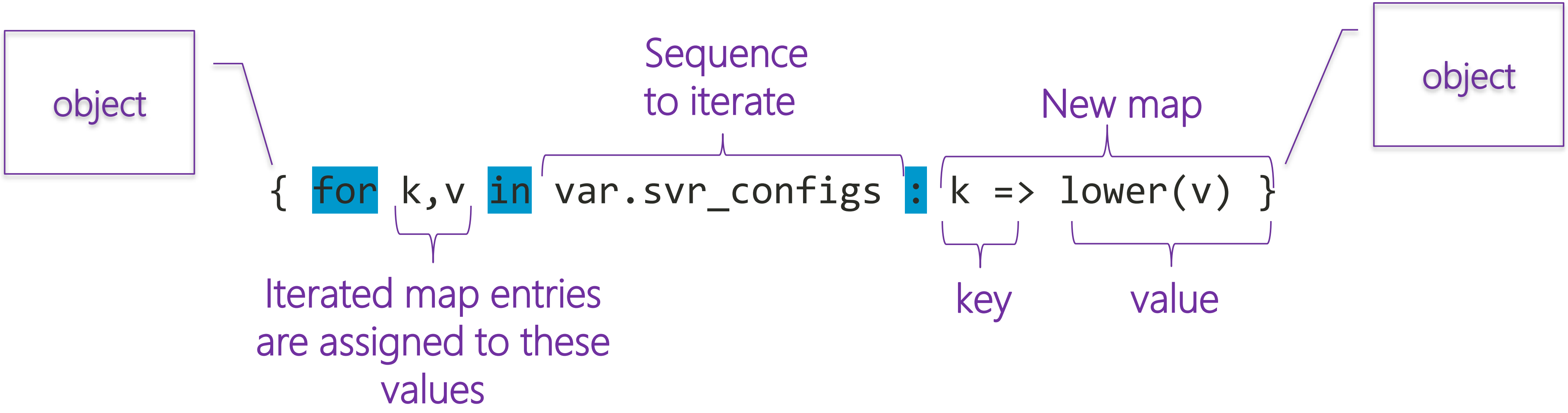
list

```
[ for s in [ "Ubuntu", "Windows", "CentOS" ] : lower(s)  ]
```

Single element is assigned to this value

Expression to perform on each element

“For” Expressions – cont’d



```
{ for k,v in var.svr_configs : k => lower(v) }
```

object

Sequence to iterate

New map

object

Iterated map entries are assigned to these values

key

value

“For” Expressions – cont’d

```
{ for k,v in var.svr_configs : k => v }

[ for s in v

{ for k,v ir                                              s) ] }
```

k	v
os	Ubuntu, windows, ...

v	s
Ubuntu, Windows, ...	ubuntu, windows, ...
1xCPU-1GB, 1xCPU-2GB	1xcpu-1gb, 1xcpu-2gb
0, 8, 16, 32, 64	0, 8, 16, 32, 64

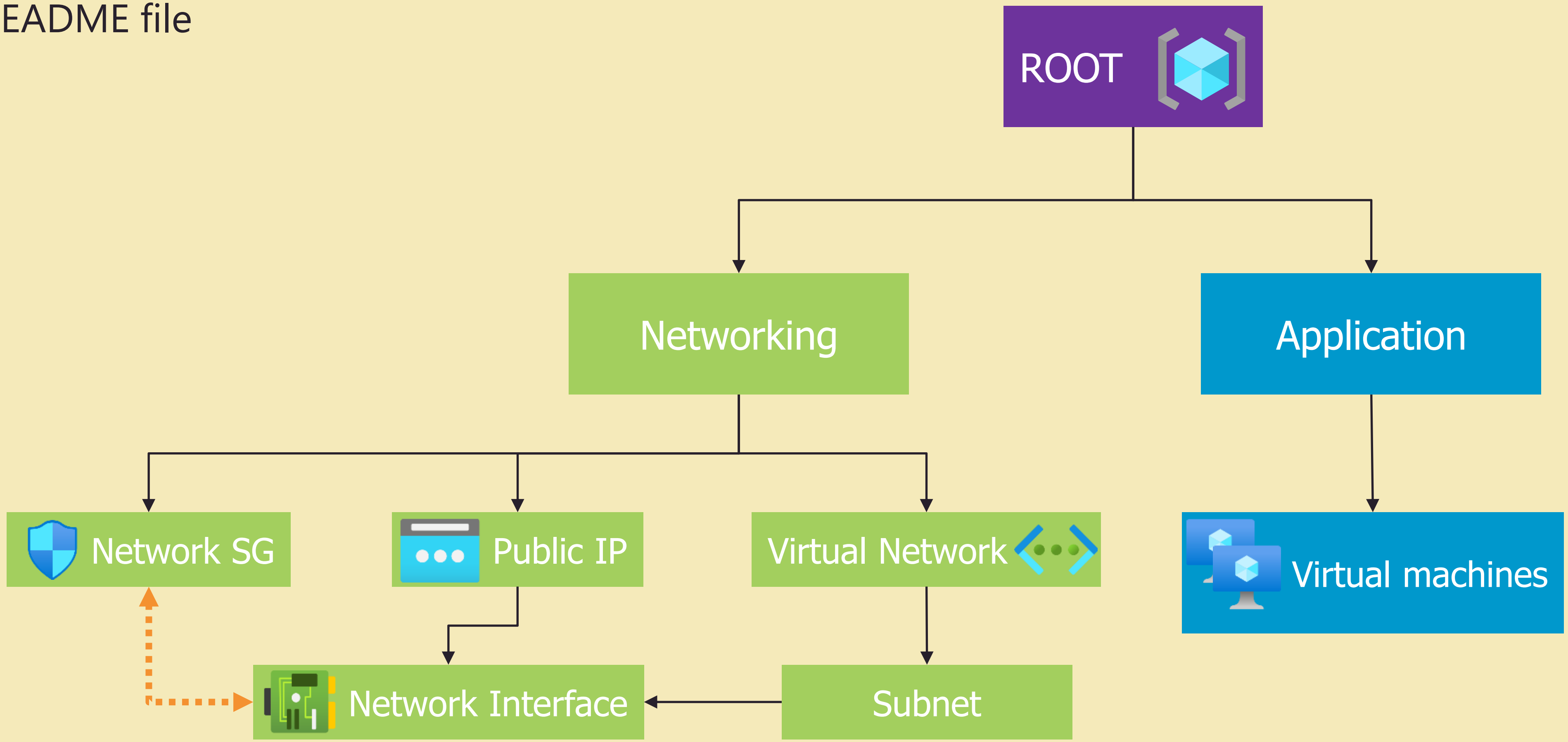
For-each + each.key/value

- For_each can be defined within a resource configuration
  - Accepts as input either a map, or a set of strings
  - Outputs an instance for each entry in the data structure
- Benefits
  - Intuitive
  - Less verbose
  - Ease of use

```
resource “azurerm_public_ip” “training” {
      for_each = toset(var.listOfMachines)
      name = “vm-${each.value}”
      …
}
…
azurerm_public_ip.training[each.key].id
…
```

Exercise 4.07 – For each virtual machine

Check README file



Cloud_init_config

- Fresh install but still no software provisioning
- cloud-init as a shell script

```
cloudinit = {
 source = "hashicorp/cloudinit"
 version = "~> 2.1"
}
```

Exercise 4.08 – Configure deployed server

Check README file

Advanced config (.yaml)

```
#cloud-config
write_files:
 -    path: /etc/server.conf
      owner: root:root
      permissions: "0644"
      content: |
      {
            "user": "${user}",
            "password": "${password}",
            "database": "${database}",
            "netloc": "${hostname}:${port}"
      }

runcmd:
 - curl -sL https://.../releases/latest | jq –r ".assets[].browser download url" | wget –qi –
 - unzip deployment.zip
 - ./deployment/server
…
```

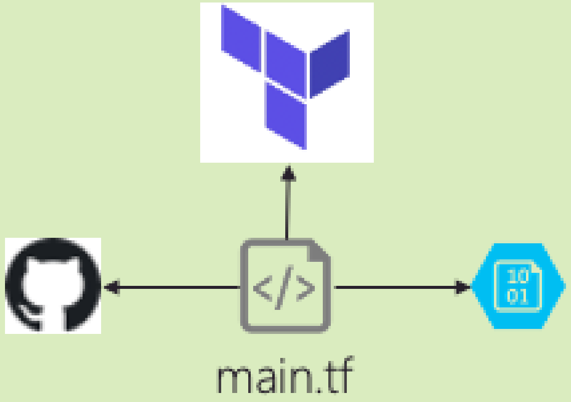
Advanced config (.yaml) – cont’d

```
data "cloudinit_config" "config" {
  gzip = true
  base64_encode = true

  part {
    content_type = "text/cloud-config"
    content = templatefile("${path.module}/config.yaml", var.infra_config)
  }
}
```

So far

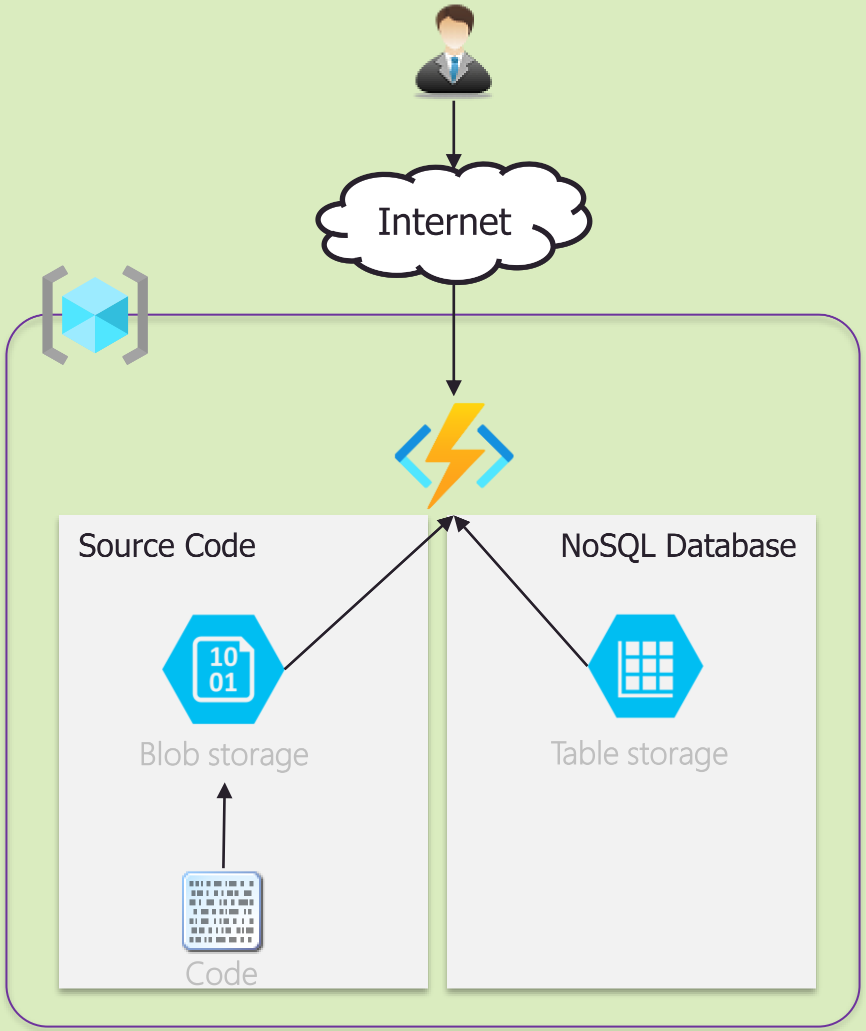
- Complex projects are RELATIVELY easy to design & deploy thanks to TF modules
- Root module is the main entry point
- Nested modules is the practice of organizing code
- Public module registry
- Data passed using a bubble-up and trickle-down techniques

Example 4.09 – Software Componentization

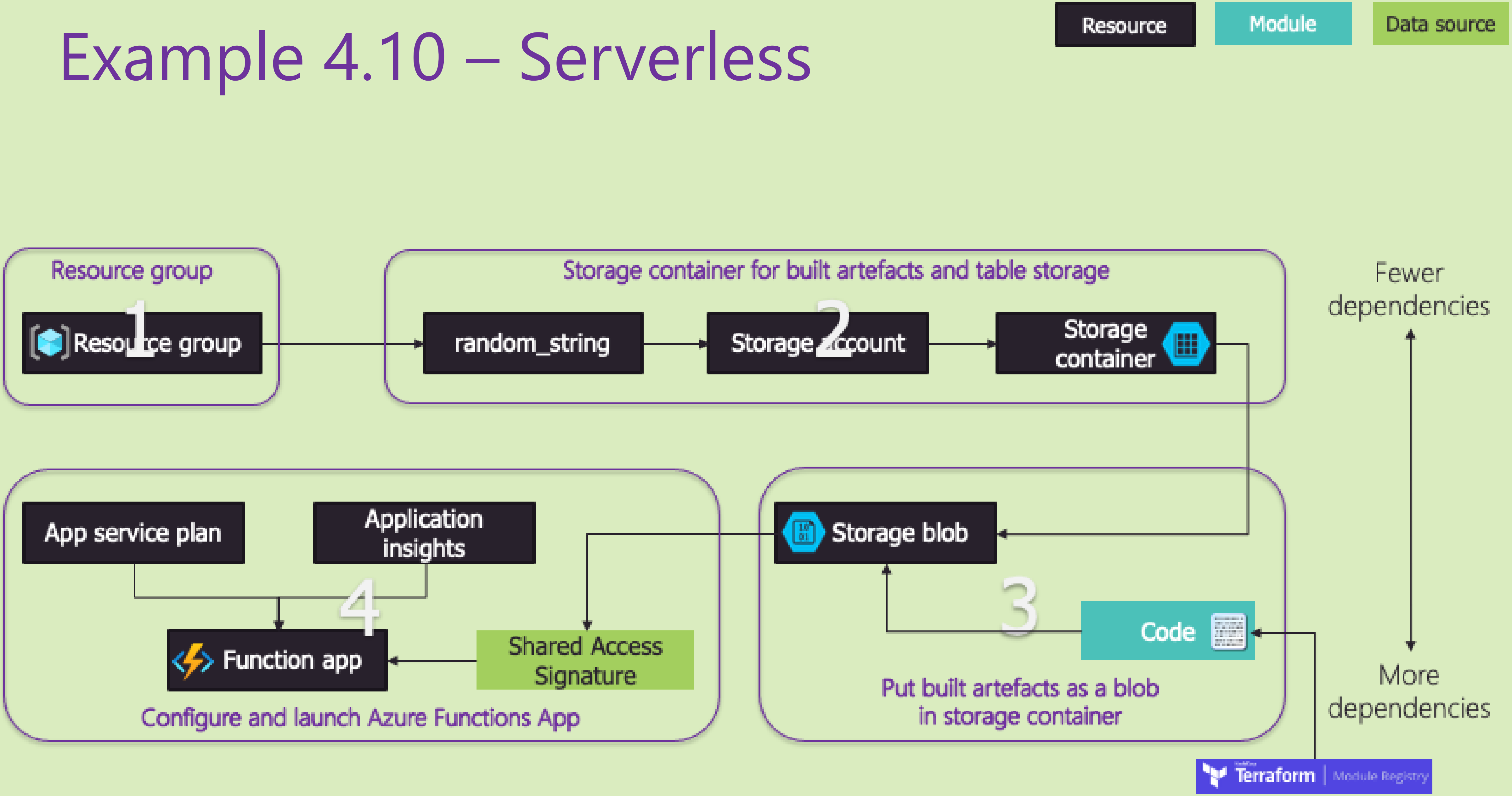


- Source code
  - [Github.com]
- Shared module
  - [Terraform Registry]
- Main.tf
  - module “tweetish”

Example 4.10 – Serverless



Example 4.10 – Serverless



Note: Groups matter more than size

- No more than a few hundred lines of code per TF file
- Grouping resources that belong together
- Organize your code in a sensible manner (NSS*)

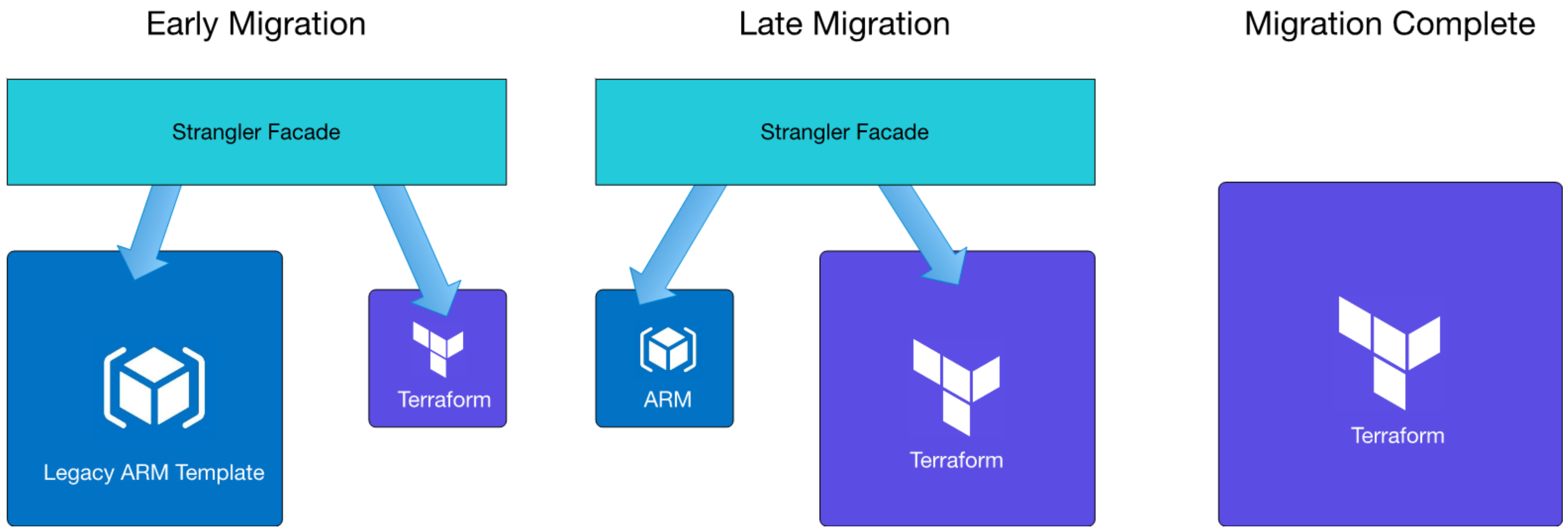
ARM and Terraform

- Legacy use cases where ARM is still useful
  - Deploying resources that aren’t yet supported by Terraform
  - Migrating legacy ARM code to Terraform
  - Generating configuration code

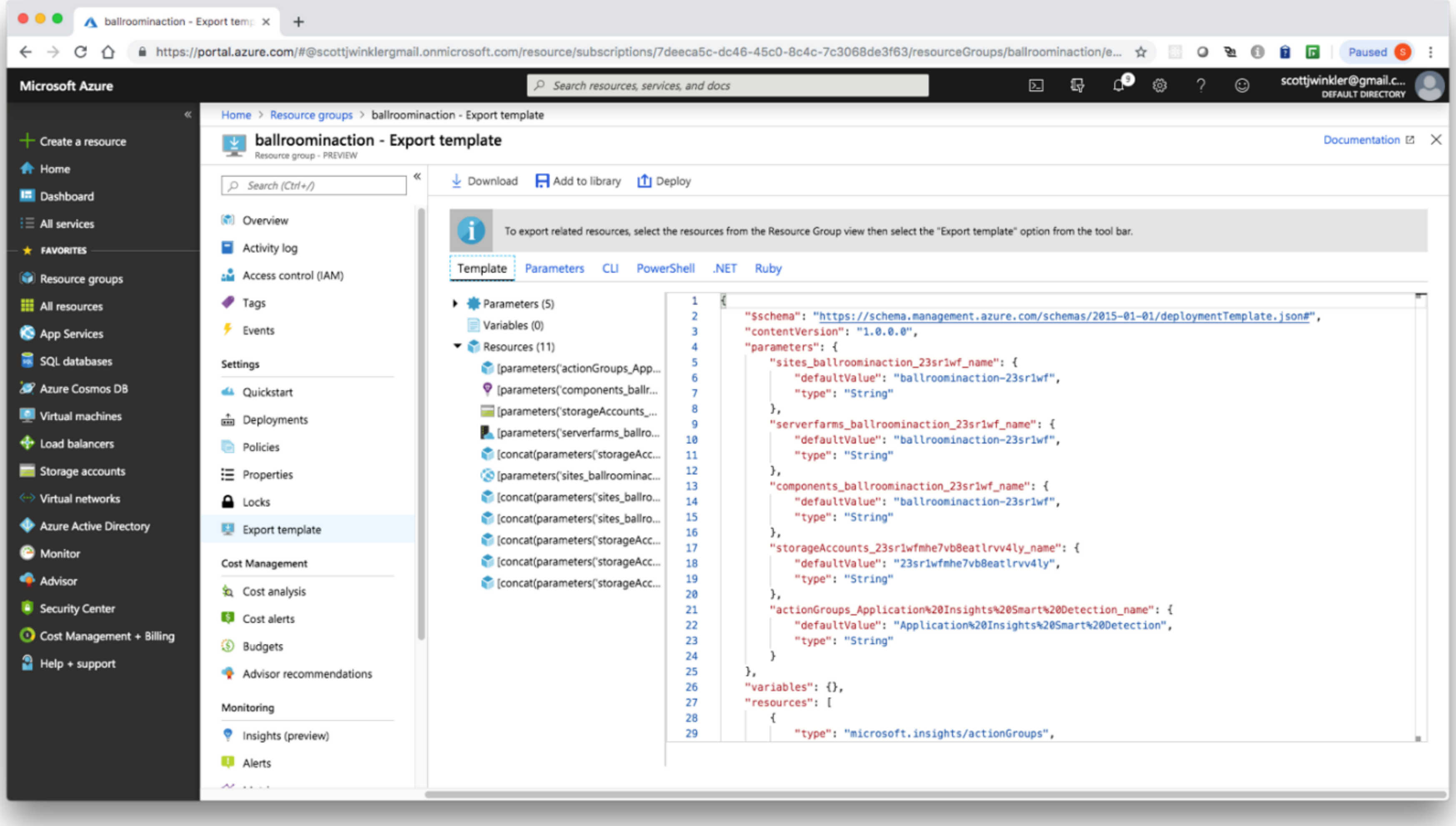
```
resource "azurerm_template_deployment" "extension" {
  name = "extension"
  resource_group_name = azurerm_resource_group.rg-app.name
  template_body = file("ARM_siteExtension.json")

  parameters = {
    appserviceName = azurerm_app_service.app.name
    extensionName = "AspNetCoreRuntime.2.2.x64"
    extensionVersion = "2.2.0-preview3-35497"
  }

  deployment_mode = "Incremental"
}
```

Migrating from Legacy Code

Strangler façade



Generating Configuration Code


ballroominaction - Export temp
https://portal.azure.com/#@scottjwinklergmail.onmicrosoft.com/resource/subscriptions/7deeca5c-dc46-45c0-8c4c-7c3068de3f63/resourceGroups/ballroominaction/e...
Paused
Microsoft Azure
Search resources, services, and docs
scottjwinkler@gmail.c...
DEFAULT DIRECTORY
Create a resource
Home
Dashboard
All services
FAVORITES
Resource groups
All resources
App Services
SQL databases
Azure Cosmos DB
Virtual machines
Load balancers
Storage accounts
Virtual networks
Azure Active Directory
Monitor
Advisor
Security Center
Cost Management + Billing
Help + support
Home > Resource groups > ballroominaction - Export template
ballroominaction - Export template
Resource group - PREVIEW
Documentation
Search (Ctrl+/)
Overview
Activity log
Access control (IAM)
Tags
Events
Settings
Quickstart
Deployments
Policies
Properties
Locks
Export template
Cost Management
Cost analysis
Cost alerts
Budgets
Advisor recommendations
Monitoring
Insights (preview)
Alerts
Download
Add to library
Deploy
Template
Parameters
CLI
PowerShell
.NET
Ruby
Parameters (5)
Variables (0)
Resources (11)
[parameters('actionGroups_App...
[parameters('components_ballr...
[parameters('storageAccounts_...
[parameters('serverfarms_ballro...
[concat(parameters('storageAcc...
[parameters('sites_ballroominac...
[concat(parameters('sites_ballro...
[concat(parameters('sites_ballro...
[concat(parameters('storageAcc...
[concat(parameters('storageAcc...
[concat(parameters('storageAcc...

To export related resources, select the resources from the Resource Group view then select the "Export template" option from the tool bar.

```json
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "sites_ballroominaction_23sr1wf_name": {
            "defaultValue": "ballroominaction-23sr1wf",
            "type": "String"
        },
        "serverfarms_ballroominaction_23sr1wf_name": {
            "defaultValue": "ballroominaction-23sr1wf",
            "type": "String"
        },
        "components_ballroominaction_23sr1wf_name": {
            "defaultValue": "ballroominaction-23sr1wf",
            "type": "String"
        },
        "storageAccounts_23sr1wfmhe7vb8eatlrvv4ly_name": {
            "defaultValue": "23sr1wfmhe7vb8eatlrvv4ly",
            "type": "String"
        },
        "actionGroups_Application%20Insights%20Smart%20Detection_name": {
            "defaultValue": "Application%20Insights%20Smart%20Detection",
            "type": "String"
        }
    },
    "variables": {},
    "resources": [
        {
            "type": "microsoft.insights/actionGroups",
```

Summary

- Terraform orchestrates serverless deployments with ease.
- Code organization is paramount when designing Terraform modules.
- Any files that are in a Terraform module are downloaded as part of terraform init or terraform get.
- Azure Resource Manager (ARM) is an interesting technology that can be combined with Terraform to patch holes in Terraform

5. Collaborating with Terraform

Content

Developing a remote backend module

Publishing modules via GitHub and the Terraform Module Registry

Switching between workspaces with the greatest of ease

Terraform states

- Why not version control ?
  - Manual error
  - Locking
  - Secrets

- Remote state = backend

Backend

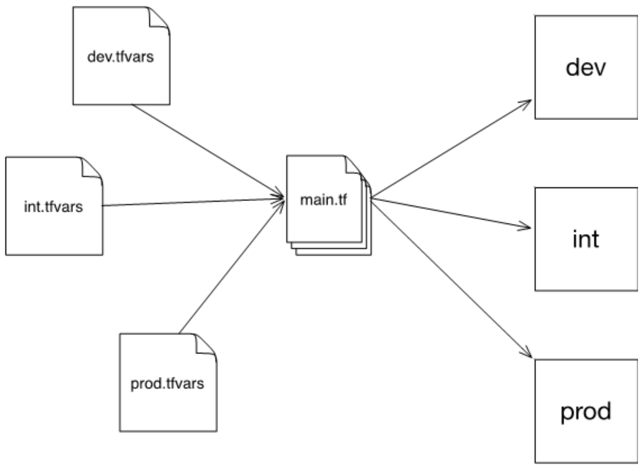
- Definition
- Tasks:
  - Synchronize access to state files via locking
  - Store sensitive information securely
  - Keep a history of all state file revisions
  - Override CLI operations
- Standard vs Enhanced backend
- Access key can be used to increase “security”

Exercise 5.01 - Azure Storage as Terraform backend

Check README file

Workspace

- Handle different subsets of the infrastructure
- Handle more than one state file for the same configuration
- There is always a “default” workspace
- Every workspace has its own variables definition



Exercise 5.02 - Workspaces

Check README file

Summary

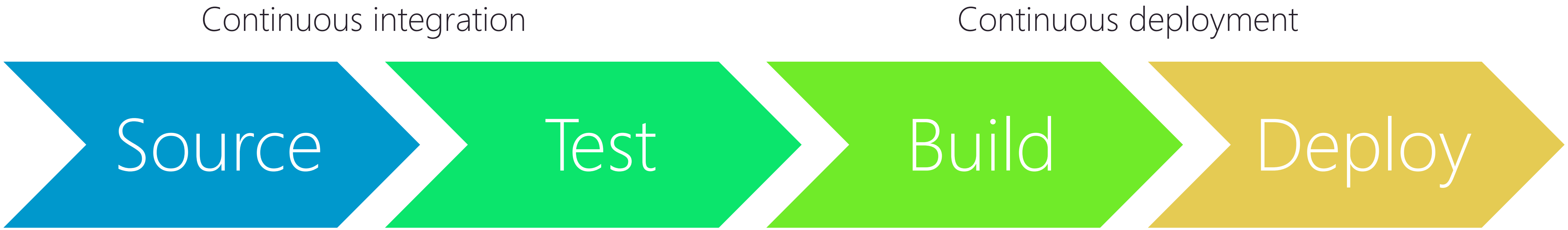
- Remote backend is probably the best option for collaboration.
- Workspaces allow you to deploy to multiple environments. The configuration code stays the same, the only thing that changes is the variables, and the state file.
- Modules can be shared through a variety of means including: Azure Storage Container, GitHub repos, and the Terraform Module Registry. You can also implement your own Private Module Registry, if you're feeling adventurous.

6. Continuous Integration/Deployment

Content

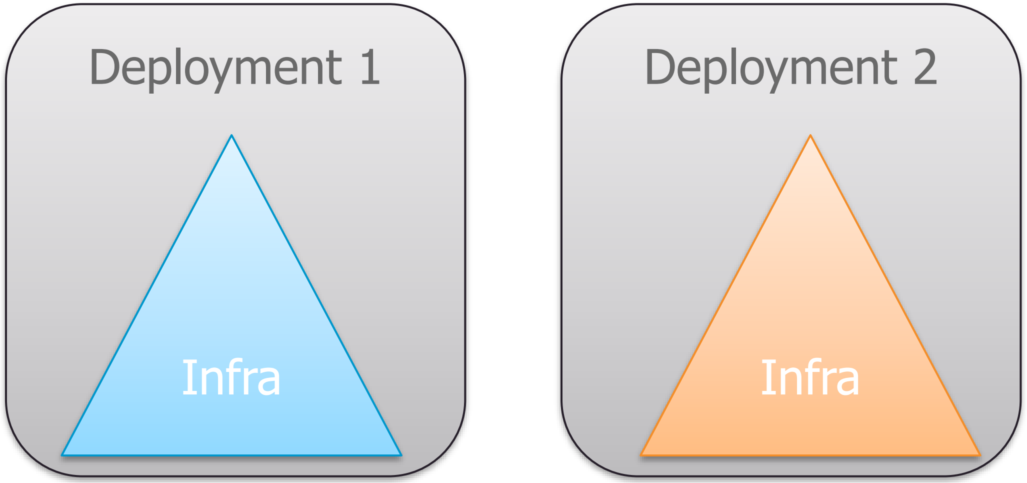
- Two-stage deployments for separating static and dynamic infrastructure
- Dynamic blocks
- Implicit vs. explicit providers and provisioners
- Use Azure DevOps to deploy a terraform-based infrastructure

CI/CD Pipeline

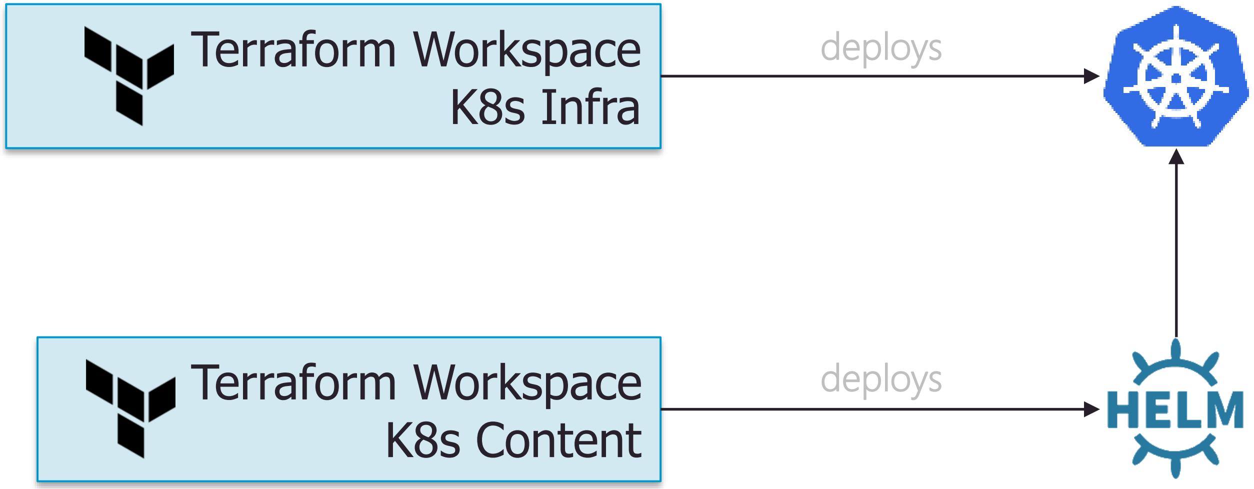


Static Infrastructure

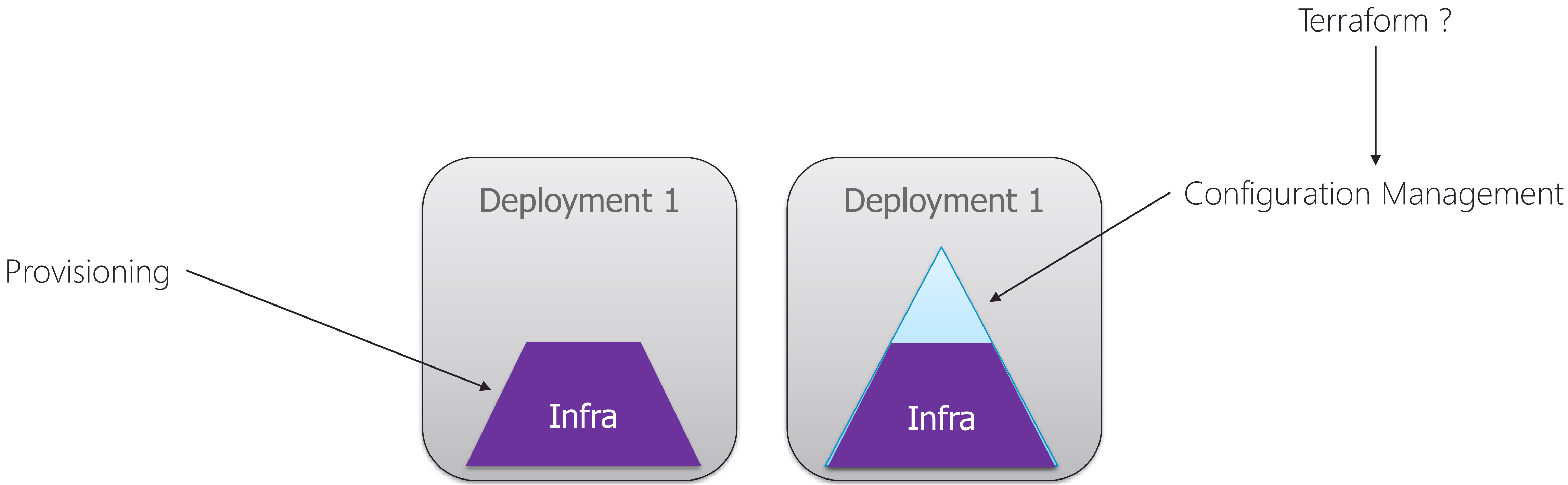
- So far,
  - Terraform not well suited for managing frequent changes
  - All-in-one deployment



What about multi-stage deployment ?



Dynamic infrastructure

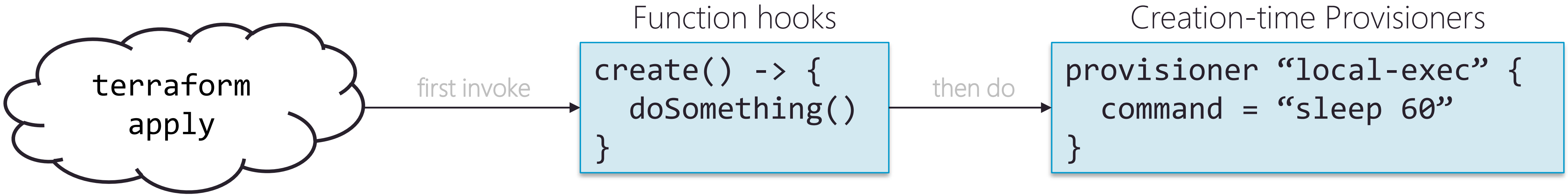


Resource Provisioner

- Race conditions - Timing is everything

- Allow to execute scripts on local or remote machine during the creation or destruction of a resource.
- Provisioner can be attached to any resource.
- There are three categories for the default provisioners: file operations, script execution, and configuration management/provisioning.

Creation-Time Provisioner



terraform apply

first invoke

Function hooks

```
create() -> {
  doSomething()
}
```

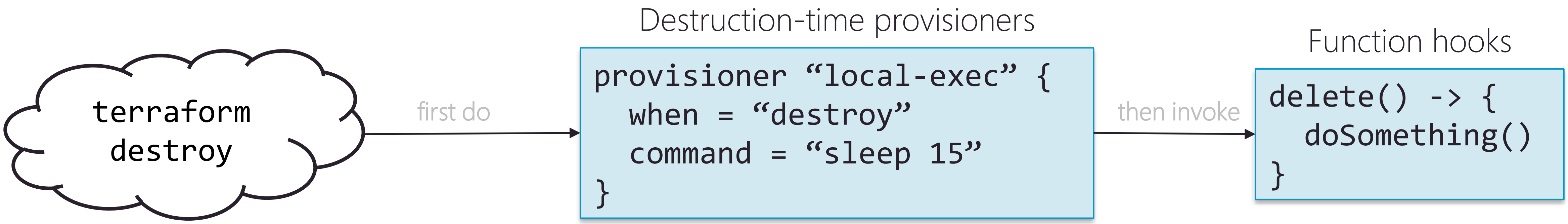
then do

Creation-time Provisioners

```
provisioner “local-exec” {
  command = “sleep 60”
}
```

Add use cases

Destruction-Time Provisioner



```
terraform
destroy
```

first do

Destruction-time provisioners

```
provisioner “local-exec” {
  when = “destroy”
  command = “sleep 15”
}
```

then invoke

Function hooks

```
delete() -> {
  doSomething()
}
```

Null resource

- File provisioner
- Script execution
  - remote-exec
  - local-exec
- Configuration Management/Provisioning
  - chef, habitat, puppet, and salt-masterless

```
resource "null_resource" "upload" {
  provisioner "file" {
    …
  }
}


resource "null_resource" "azure-cli" {
  provisioner "local-exec" {
    command = "ssl-script.sh"
  }
}
```

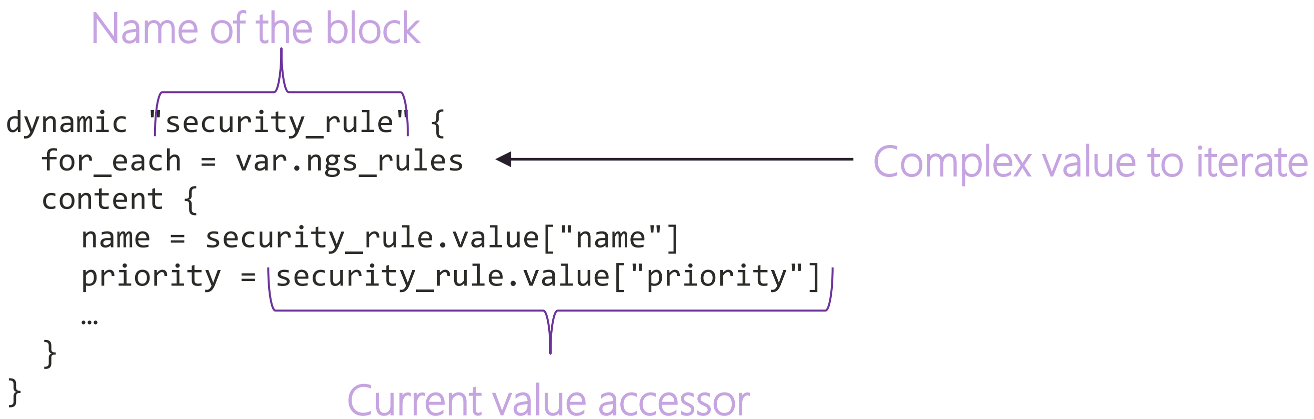
Example 6.01 – Null resource & provisioner

Remote-exec example

```
provisioner "remote-exec" {
  inline = [
    "echo \"nameserver 8.8.8.8\" | sudo tee -a /etc/resolv.conf",
    "sudo yum update -y",
    "sudo yum install epel-release -y",
    "sudo yum install puppet-agent -y",
    "sudo /opt/puppetlabs/bin/puppet agent --version"
  ]
  connection {
    user = "centos"
  }
}
```

Dynamic Blocks

- Use to dynamically create a nested configuration block;
- Can *only* be used within other blocks, and *only* when the use of repeatable configuration blocks is supported;
- See them as *for* expressions



Name of the block

```
dynamic "security_rule" {
  for_each = var.ngs_rules
  content {
    name = security_rule.value["name"]
    priority = security_rule.value["priority"]
    …
  }
}
```

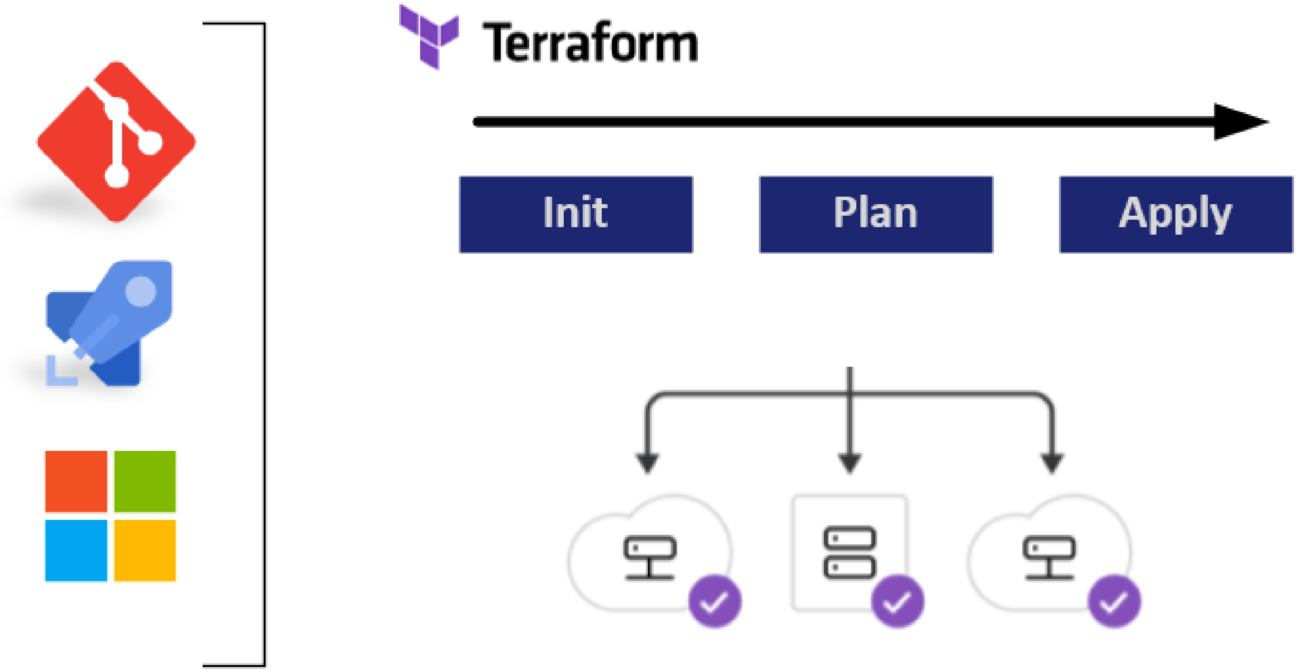
Complex value to iterate

Current value accessor

Exercise 6.02 – Dynamic block

Check README file

Azure DevOps – CI/CD



3 "requirements":

- Backend
- Azure DevOps Project
- Azure Service Principal

Exercise 6.03 – Azure DevOps & Terraform

Exercise 6.04 – Dynamic backend

Summary

- CI/CD pipeline can easily be supported by Azure DevOps
- Resource provisioner complement the Terraform runtime.
- Dynamic block is not often used by can speed up the elaboration of scripts and facilitate their reading
- Combining Azure DevOps with Terraform increase the robustness of your CI/CD methodology

7. Deployment

Content

Customizing resource lifecycles with the create_before_destroy flag

Performing Blue/Green deployments with Terraform

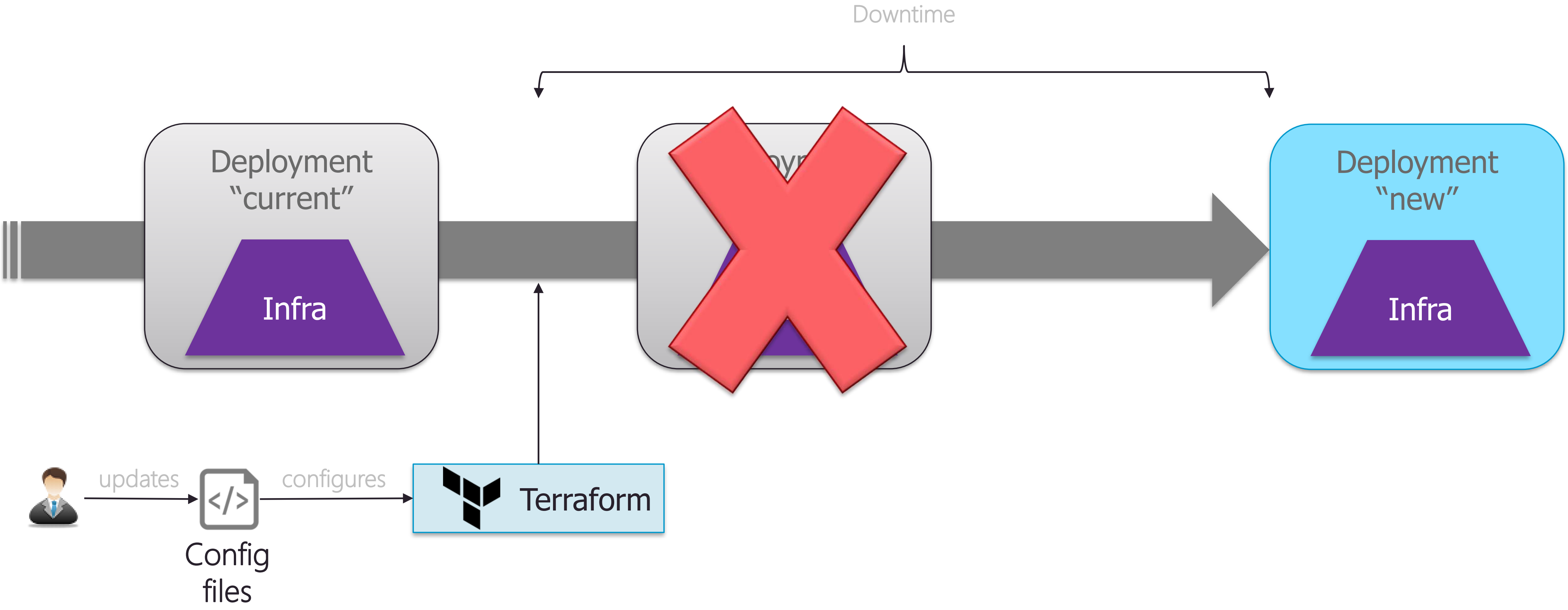
Installing software on virtual machines with remote-exec provisioners

Zero Downtime Deployment

- Definition: Practice of keeping services always running and available

- Terraform’s approaches:
  - Use “create-before-destroy” meta-attribute
  - Blue/Green deployments
  - Responsibility transfer

Lifecycle Customizations



Lifecycle meta-argument

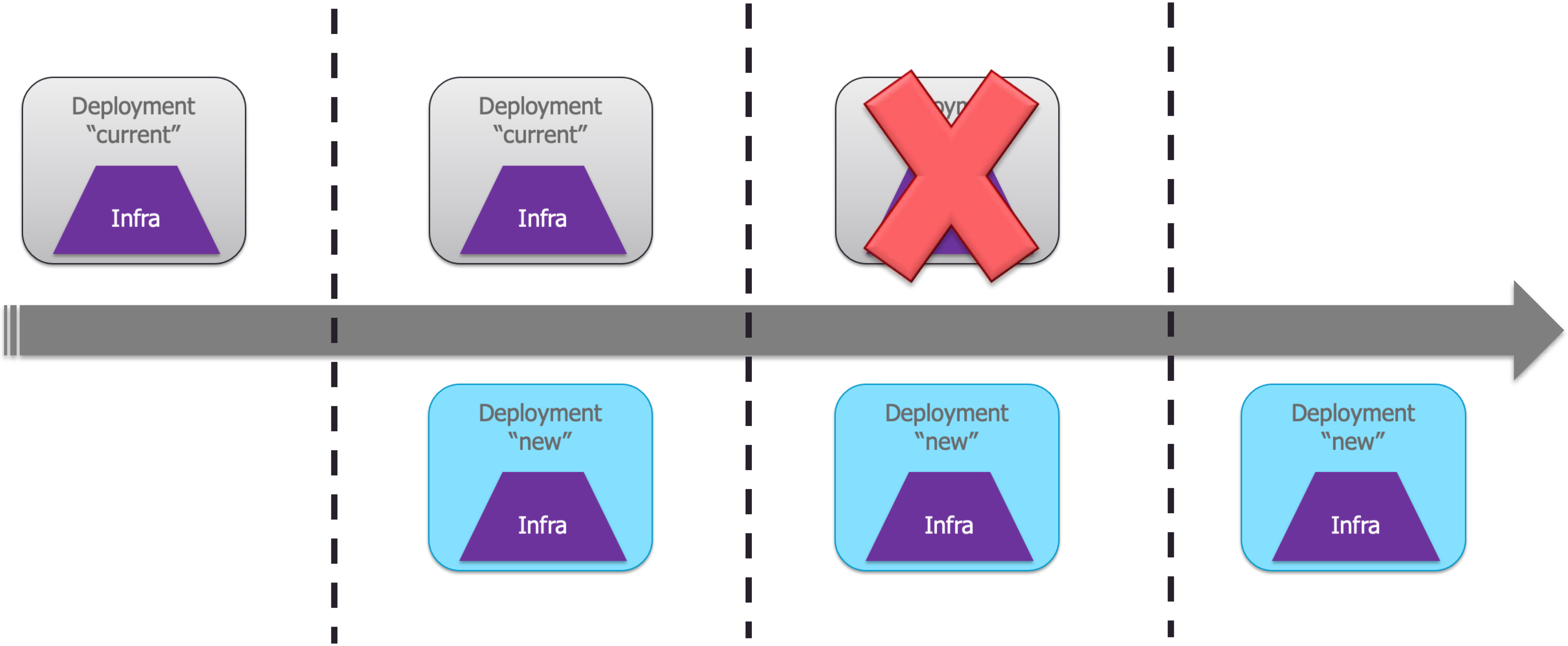
- “lifecycle” nested block in any resource

- Three flags:
  - “create_before_destroy” (bool)
  - “prevent_destroy” (bool)
  - “ignore_changes” (list of attribute names)

- Use these flags to override the default behaviour

```
resource “type” “name” {
  lifecycle {
    flag_one = …
    …
  }
}
```

Note: “create_before_destroy” flag



Considerations about “create_before_destroy”

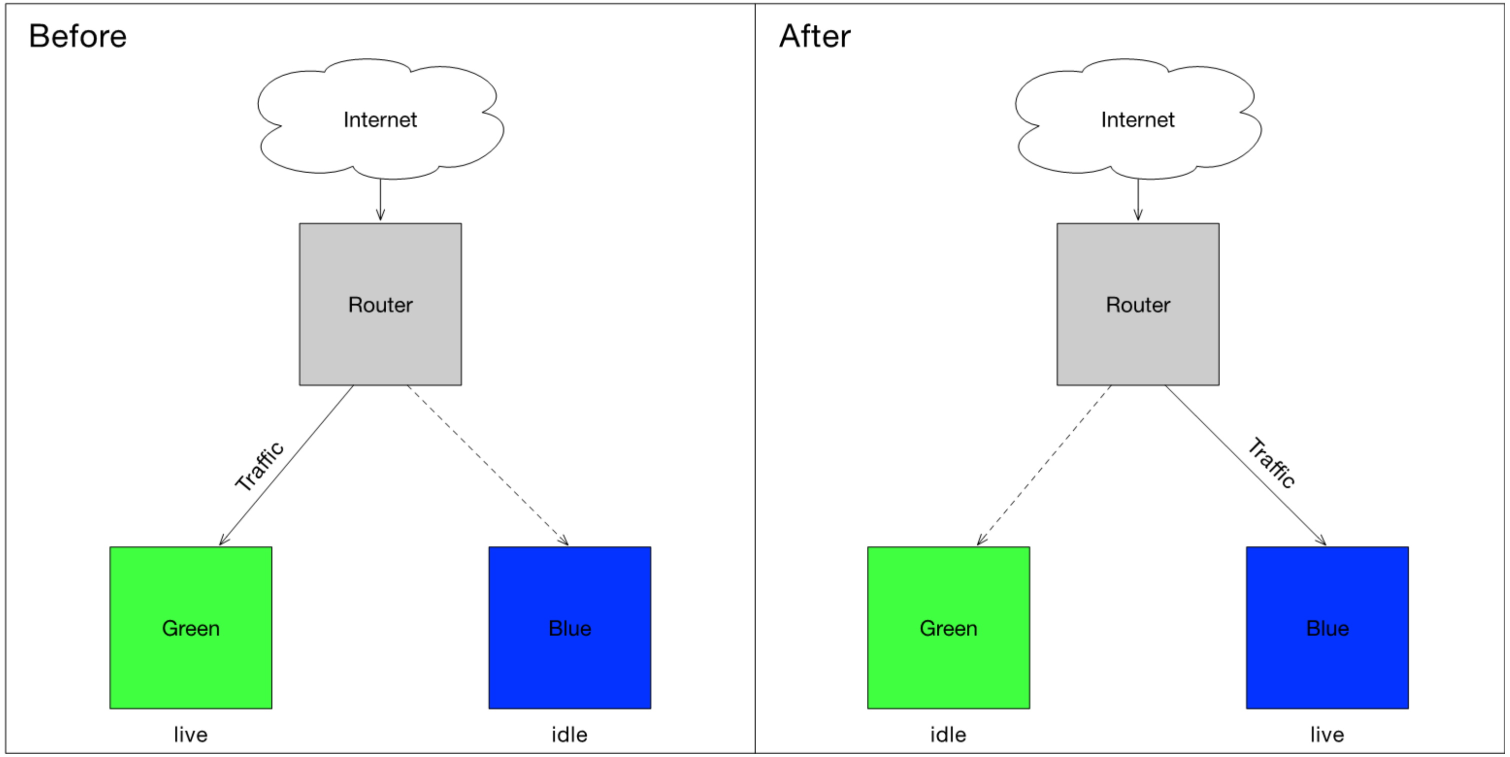
- The “create_before_destroy” flag
  - Can be confusing: affects the default behaviour of Terraform
  - Is redundant: alternatives such as workspaces or modules
  - Can create namespace collisions
  - “*Force new*” vs “*updated in-place*”

Exercise 7.01 – Lifecycle flag

Check README file

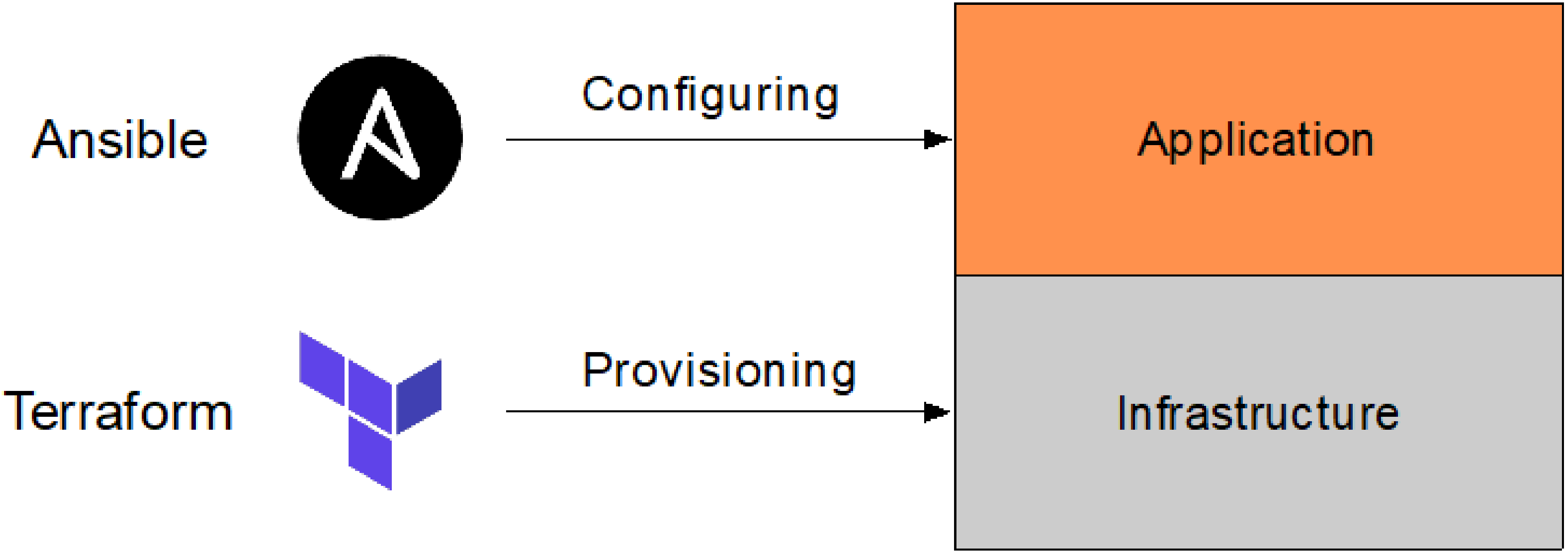
Blue/Green Deployments

- The use of modules can help but probably not the best

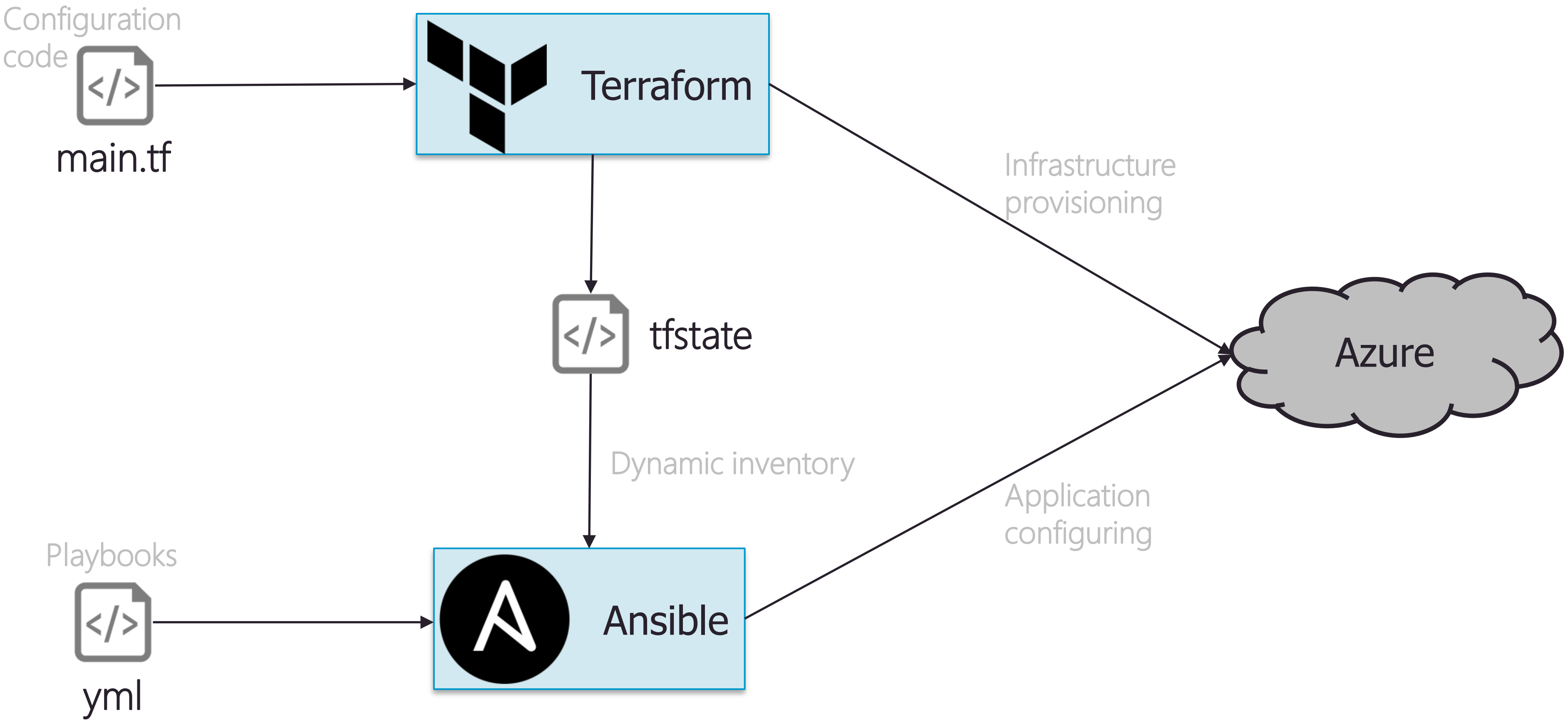


Configuration Management

- Enables rapid software delivery onto existing server



Terraform + Ansible



Exercise 7.02 – Terraform & Ansible

Check README file

Summary

- Zero Downtime Deployment (ZDD) is the practice of keeping services always running and available to customers.
- Performing Blue/Green deployments in Terraform is more a technique than a best practice.
- Combination with configuration tool should always be considered

8. Testing and Refactoring

Content

- Resource tainting
- Module expansion refactoring techniques
- Migrating state with terraform mv and terraform state commands
- Testing Infrastructure as Code with terraform-exec

Refactoring

- Definition: continuous improvement of the code design with marginal impact on behaviour

- Refactoring goes further by strengthening maintainability, increasing extensibility and facilitating reusability

Selective refactoring

- **`terraform taint`** forces a resource to be destroyed and recreated on the next “apply” command
- Tainted resources appears as such in the plan
- Resources can be untainted

- Use cases:
  - Reset a resource to its initial state
  - Force the rolling of security groups/keys
  - Partial rebuild

Exercise 8.01 & 8.02 – Rotate access keys

Check README file

Securing Terraform state

- Terraform does not treat attributes containing sensitive data any differently than it treats non-sensitive attribute.
- Three methods for securing state files:
  - Removing Unnecessary Secrets from Terraform State
  - Least Privileged Access Control
  - Encryption at Rest

Exercise 8.03 – Azure KeyVault Secrets

Refactoring – Modularizing Code

- Typical deficiencies:
  - Duplicated Code
  - Name Collisions
  - Inconsistency
- The biggest refactoring improvement we can make it to put reusable code into modules.

- Module expansions (only with TF 0.13)

Exercise 8.04 – Modularizing Code

Check README file

Terraform State Migration

- Refactoring, and particularly software componentisation (modularizing code) implies to re-initialise the workspace.
- Generating the plan will reveal that all resources will be destroyed and created during execution (apply).
- There are 3 options:
  - Manually editing the state file (not recommended)
  - Moving stateful data with terraform state mv
  - Deleting old resources with terraform state rm and reimporting with terraform import

Moving resources

- Moving an existing state from their current resource address to their final resource address.
- `terraform state mv [options] SOURCE DESTINATION`

Exercise 8.05 – Moving resources

Check README file

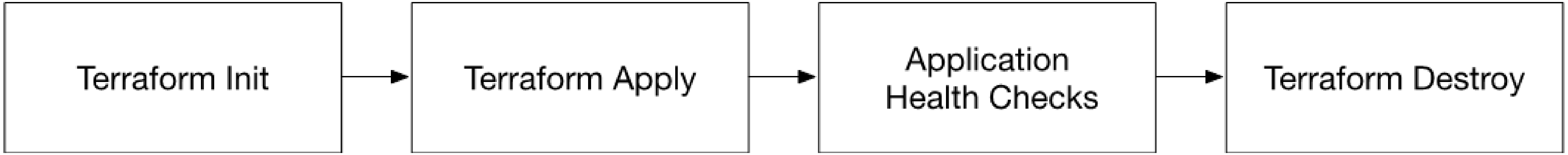
Importing Resources

- Migration can be performed through a DELETE + IMPORT process
- `terraform state rm ADDRESS`
- `Terraform import ADDRESS`

Testing

- Testing should be at least done at three levels:
  - Unit testing
  - Integration testing
  - System testing

- terraform-exec (in GO language)



Terraform Init

Terraform Apply

Application Health Checks

Terraform Destroy

Summary

- “taint” is useful to rotate/refresh time sensitive resources
- Module expansion should be favoured to flat module structure
  - Move resources / modules
- Unmanaged resources can be converted through import

Thank you !