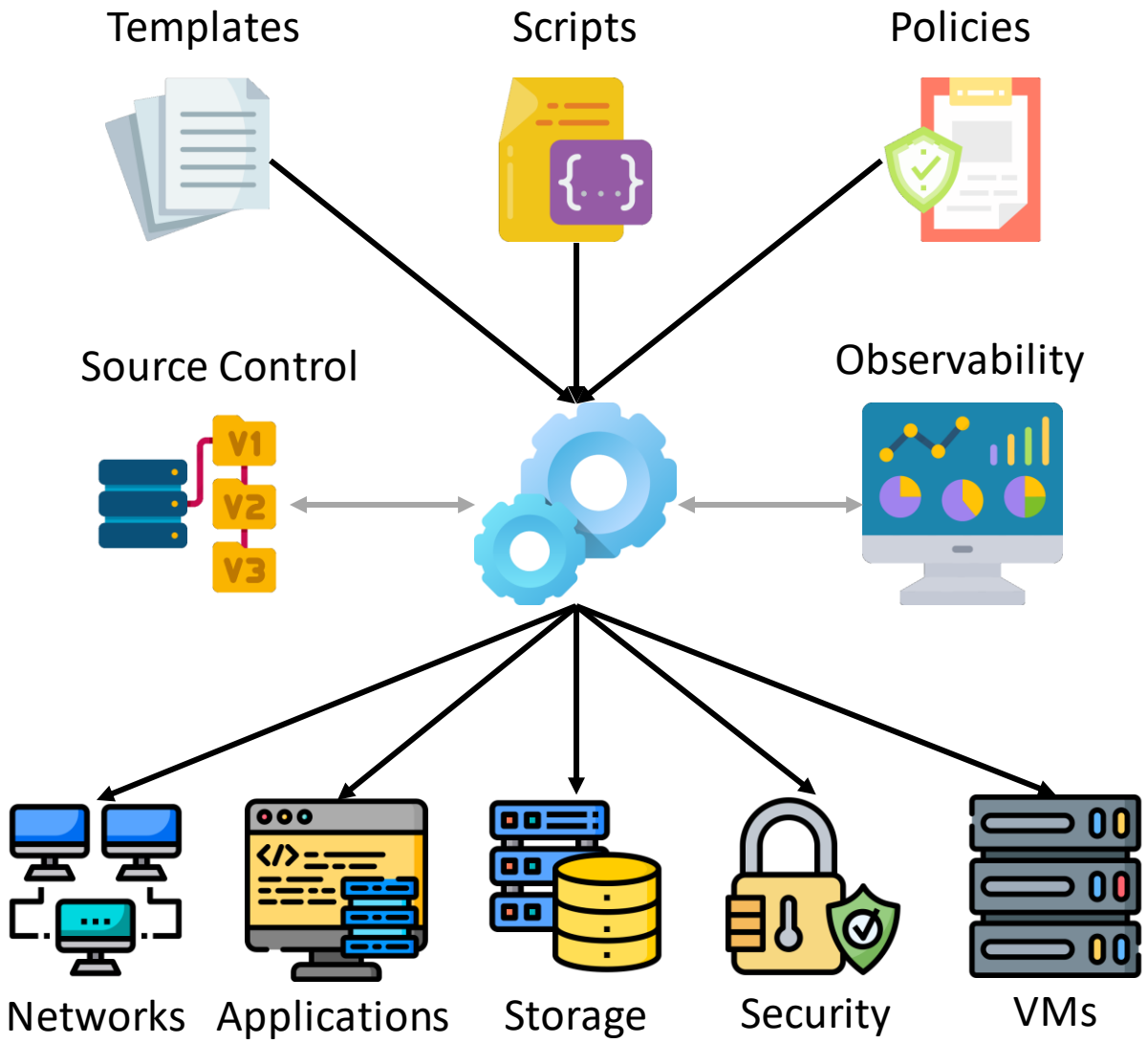
Describe and Provision Infrastructure

Infrastructure as Code

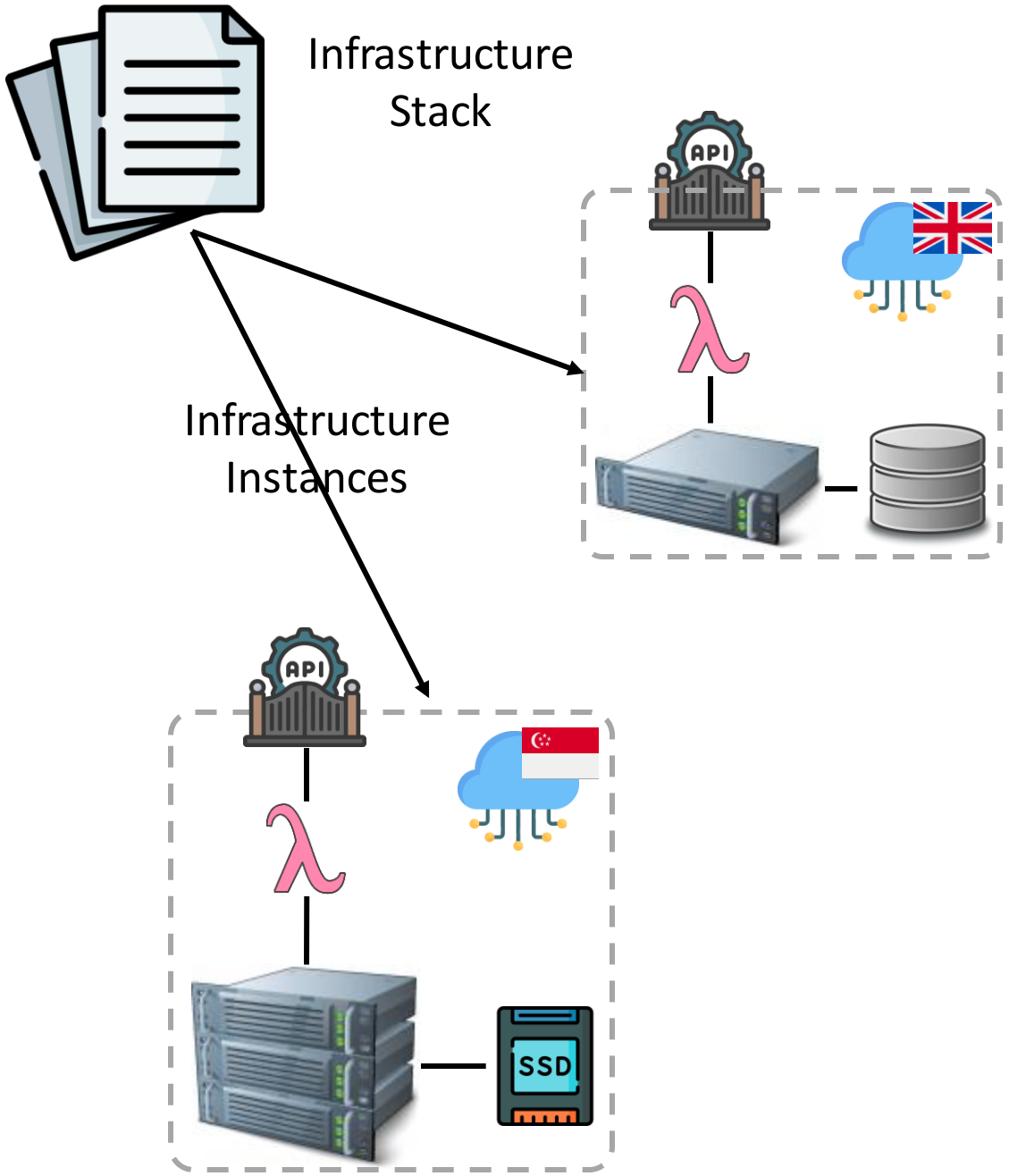


- Tool that enables the automatic provisioning of infrastructure
- Write scripts to describe and declare what that infrastructure looks like
- Integrates with development and operations workflow

Image adapted from http://expertcloudengineer.com/wp-content/uploads/2019/03/5d8bb08773acf-550x386-1.png

Infrastructure Stack

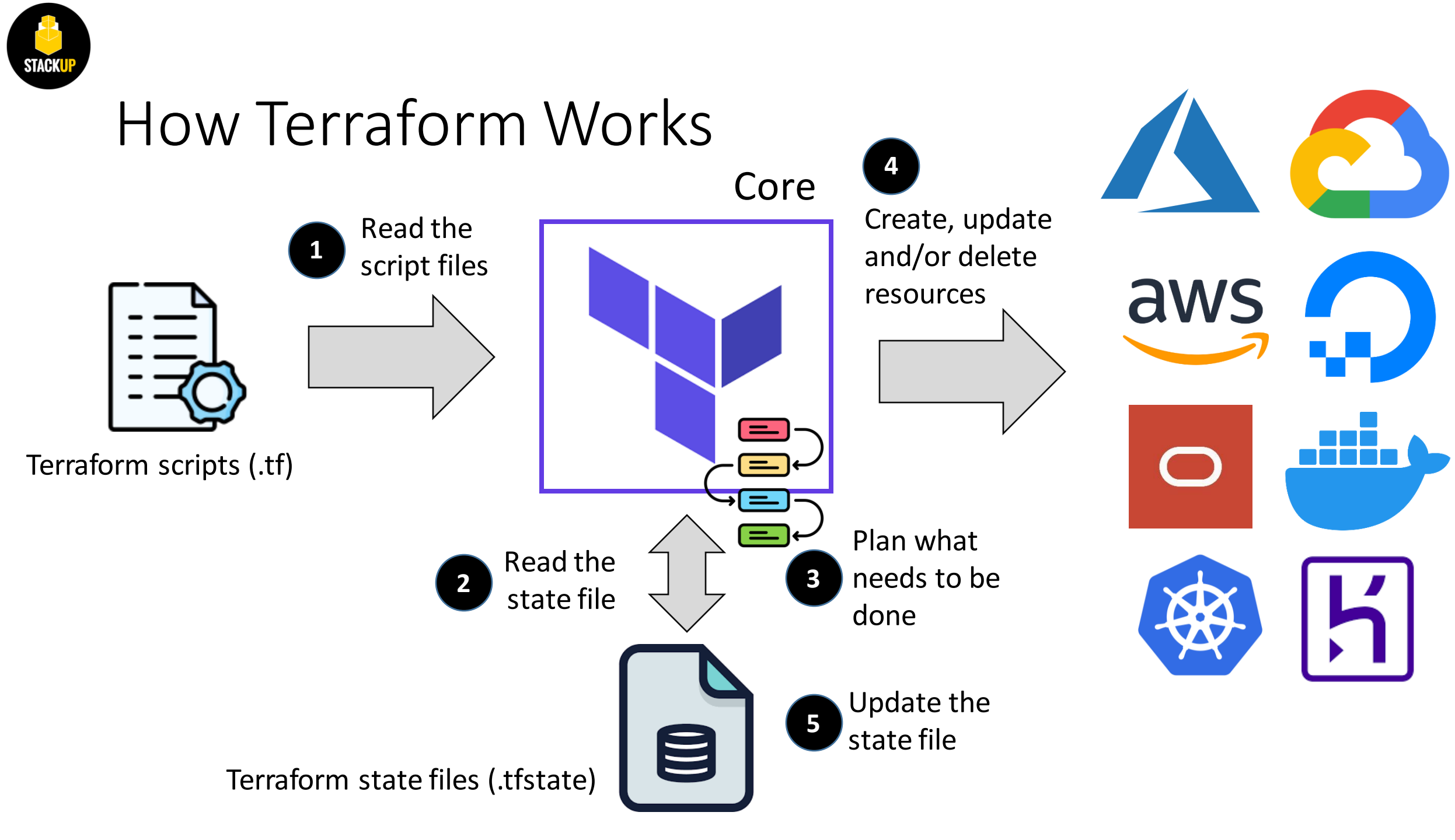
- Collection of resources that is defined, provisioned and updated as a single unit
  - Compute - eg virtual machine
  - Storage - eg disk
  - Networking - eg. VPC, routes
  - Services - eg. application, security
- Instantiate multiple stack instances from the infrastructure stack
- Infrastructure stacks can be parameterized
  - Deployed with different components
  - For different environments



What is Terraform?

- A tool to provision infrastructure cloud/virtualized resources
- Open source
- Extensible
  - Add support to new environments or resources
  - Write your own provider to integrate with any cloud platform, REST endpoint, tools
- Uses HCL (HashiCorp Configuration Language) to define and describe the infrastructure
  - Declarative
  - HCL is used in other tools
- Idempotent
  - Will only provision the required resource to make the actual the same as the desired

How Terraform Works



Key Features

- Infrastructure as code
  - Describe the layout of the infrastructure that you application requires
- Execution plans
  - Reconcile the difference between the desired (your HCL scripts) and the actual
- Resource graphs
  - Resources are dependent on each other so will have to be created in the correct order
  - A graph is created for these dependencies
  - When provisioning the resources, which resources have to be serialized and the creation of which resources can be parallelized
- Change automation
  - Automate and manage the rollout and teardown of resources with the execution plan and resource graphs

Terraform Core Concepts - 1

- Providers
  - ‘Drivers’ for different infrastructure providers eg. AWS, Docker, Heroku
  - Providers for IaaS, PaaS, SaaS
  - Open architecture, implemented in Golang
  - https://registry.terraform.io/browse/providers
- Resources
  - Description and configuration of resources on the infrastructure provider
    - Eg. virtual machines, CI/CD pipelines, databases, firewall, API gateway
  - Resources are tied to providers
    - Cannot provision resource from one provider on another provider’s infrastructure
    - Resources are not ‘cross platform’

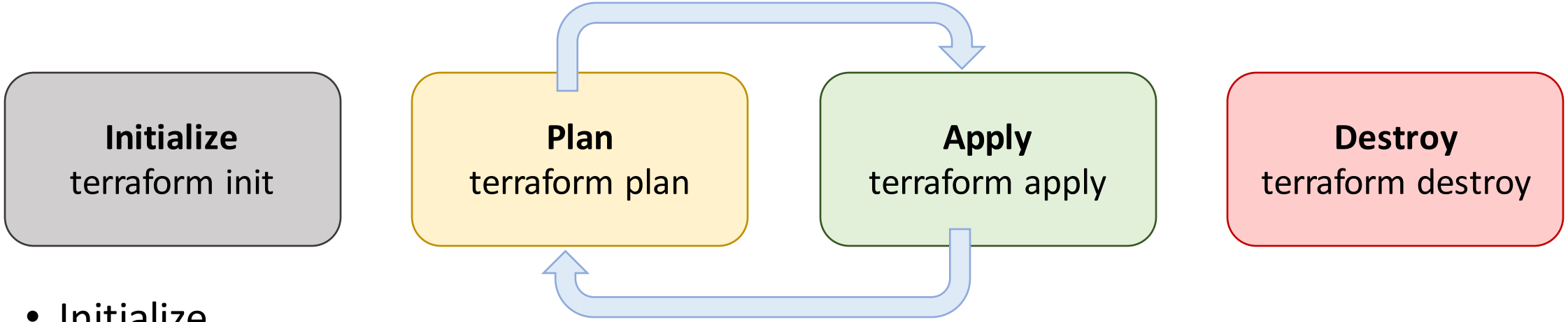
Terraform Core Concepts - 2

- Data source
  - Information fetched from an external source or from the infrastructure provider itself
  - Eg. name of all the regions from a cloud provider
  - Eg. get a list of all the provisioned VMs (not necessarily managed by Terraform)
- State
  - Represents what Terraform knows (or think it knows) about the infrastructure
  - Holds information and configuration of the rollout of the scripts
  - Uses this file to reconcile the difference between the actual and the desired

Terraform Core Concepts - 3

- Input Variables
  - Configuration parameters for the scripts
    - Eg. API key, region, VM size, image name
  - Variables are typed
- Output Values
  - Results from executing the scripts
    - Eg. IP address of the newly created virtual machine
  - Can be used with resources from other providers
    - Eg Add an A record with the IP address from the newly provision droplet on DigitalOcean to GoDaddy
    - Eg. Use the AWS RDS endpoint in GCP's AppEngine

Terraform Workflow

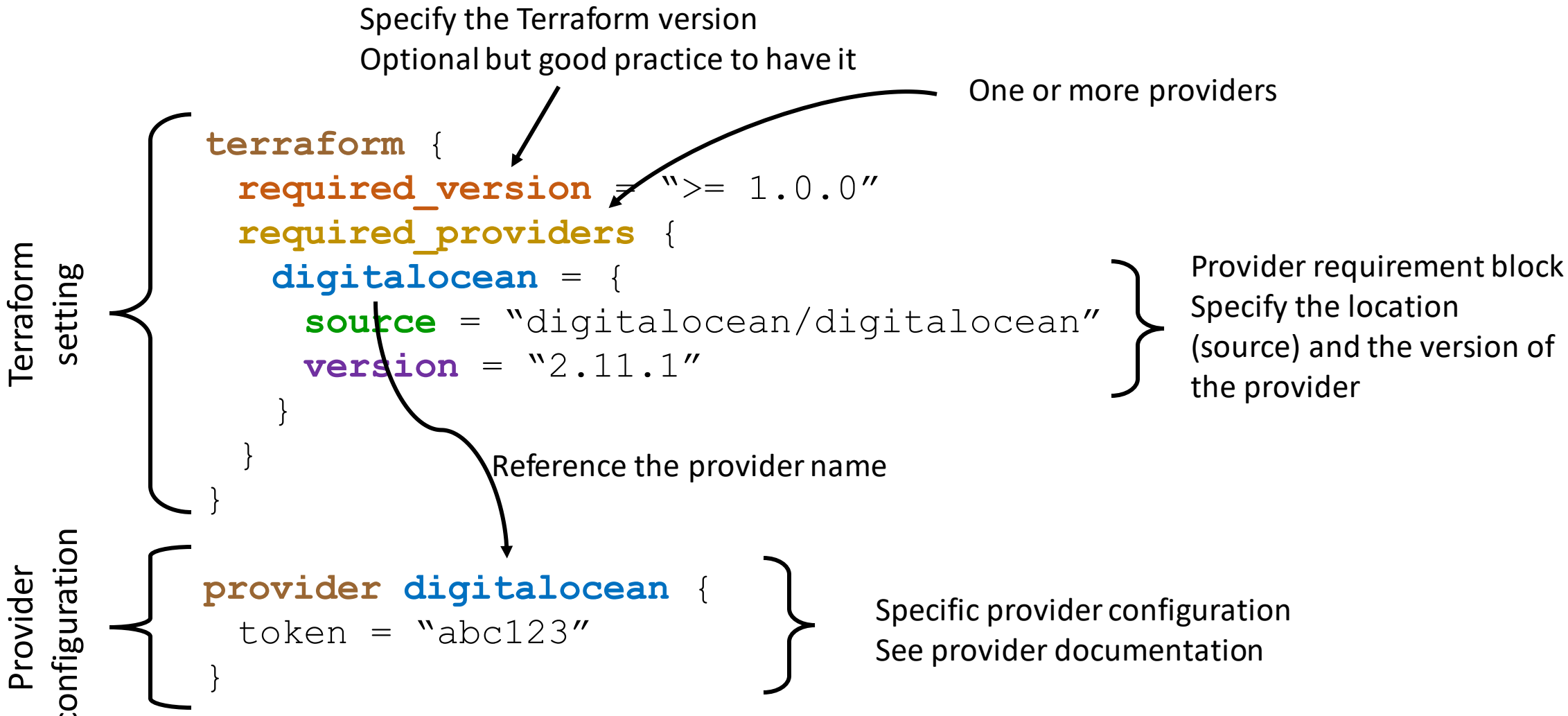


- Initialize
  - Initializes the project directory, download and install providers, create configuration files
  - Performed whenever a new provider is added
- Plan
  - Analyze the state of the infrastructure and create an execution plan
- Apply
  - Make changes to the infrastructure as per execution plan
- Destroy
  - Destroy the provisioned resources

Provider

- Providers are ‘drivers’ that integrates with specific platform
  - Run once at the start your project
  - Every provider is configured differently
  - See https://registry.terraform.io/browse/providers
- Provider are downloaded when `terraform init` is executed
  - Need to rerun `init` if new providers are added to the project
- Providers and Terraform project are configured with terraform setting configuration block
  - https://www.terraform.io/docs/language/settings/index.html

Configuring Provider



Example - Provider

```
terraform {
  required_version = “>= 0.15.0”
  required_providers {
    aws = {
      source = hashicorp/aws “”
      version = “3.37.0”
    }
  }
}

provider aws {
  region = “ap-southeast-1”
  access_key = “abc123”
  secret_key = “xyz789”
}
```

```
terraform {
  required_version = “>= 0.15.0”
  required_providers {
    docker = {
      source = "kreuzwerker/docker"
      version = "2.15.0
    }
  }
}

provider docker {
  host = “tcp://192.168.0.10:2376”

  cert_path = pathexpand(
     “~/.docker/machine/machinies/mydocker”)
}
```

Data Source

- Fetch data/information from external resources
  - Like making a RESTful API call
  - These resources are not managed by your Terraform project
- Use cases
  - Get the list of available regions
  - Find a specific image for the cloud provider's store based on some criteria
- Data sources are prefixed with `data`

Example - Data Sources

```
data digitalocean_ssh_key mykey {
  name = "mykey"
}

data digitalocean_droplets webapp {
  filter {
    key = "regions"
    values = [ "sgp1" ]
  }
  filter {
    key = "tags"
    values = [ "eng", "web", "v1" ]
    all = true
  }
}
```

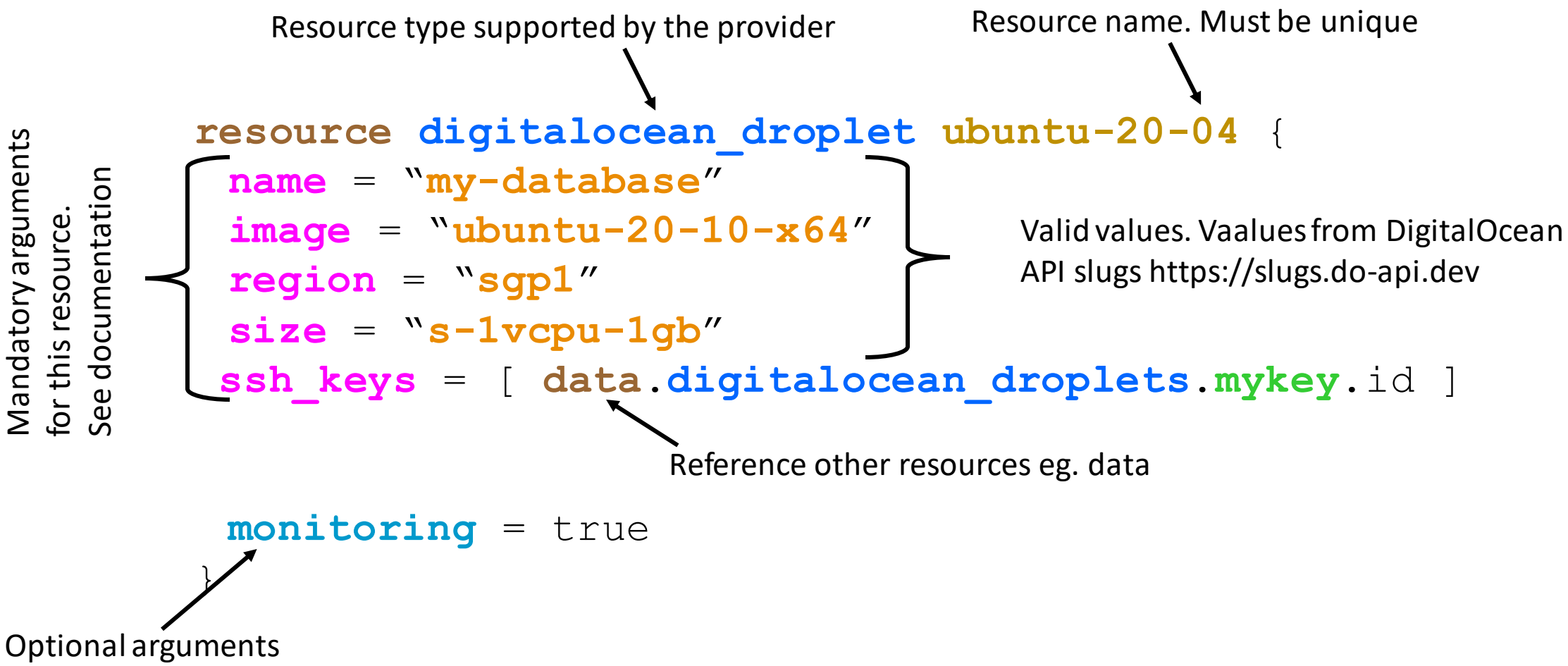
Lookup a SSH key from the cloud provider and add it to the droplet

Filter droplets by region and tags

Resource

- Resources are specific to the configured providers
- Resource definition consist of
  - Resource type - eg. `docker_container`, `aws_ebs_volume`, `digitalocean_loadbalancer`
  - Note: the first word in the resource name is the provider's name
  - Resource name
    - This is like a class name, when we provision from this resource, every instance must have a unique name but the all share this same resource name
    - Eg. `myapp` vs `myapp-01`, `myapp-02`, `myapp-03`
  - Arguments to configure the resource
    - Some are mandatory eg, name, some are optional
    - Depends on the resource type

Example - Resource



Referencing Terraform Objects

- Resources hold values which can be reference by other objects
  - Eg. Add IP address of a provisioned VM to an API gateway
  - See ‘Attribute Reference’ in documentation
- Resources can be referenced by their named value with the ‘dotted’ notation
  - Resources - `<resource_type>.<name>`
    - Eg. `digitalocean_droplet.ubuntu_20_04.ipv4_address`
  - Input variables - `var.<variable_name>`
  - Data sources - `data.<data_type><name>`
  - Module - `module.<module_name>`
  - Local variables - `local.<variable_name>`

Provisioning and Destroying the Stack

- `terraform apply` reports what are the changes it'll make to the stack
  - Need to confirm before proceeding
  - Produces a state file - `terraform.tfstate`
- `terraform destroy` destroys the stack instance
  - All provision resources will be destroyed
- Both operations require confirmation before proceeding
  - Type yes
  - Use `-auto-approve` option to skip manually typing yes

Values

- Input variables - passed to Terraform in `apply` or `plan` operations
  - Typed
  - Typically used to configure the stack
  - Eg. keys, region, volume size, etc.
- Output values - attribute values of resources after an `apply` operation
  - Values will only be available after resources have been provisioned
    - Will see ‘known after apply’ during a plan
  - Read only
  - Eg. IP address, fingerprint of a SSH key, resource ids

Declaring Input Variables

```
variable DO_key {
  type = string
}

variable deploy_region {
  type = string
  default = "sgp1"
}

variable monitor {
  type = bool
  description = "Enable monitoring"
  default = false
}

variable comments { }
```

- Input variables are typed
  - primitive - string, number, bool
  - complex - list, map, object, tuple
- If no type is specified, then variable can accept any type
  - Good practices to declare the type
- Variable arguments, all are optional
  - `type` - variable type
  - `default` - default value
  - `description` - description of the variable
  - `validation` - rules to validate the value

Example - Variables

```
variable droplet_region {
  type = string
  default = “sgp1”
}

variable droplet_size {
  type = string
}

resource digitalocean_droplet ubuntu_20_04 {
  name = “my-database”
  image = “ubuntu-20-10-x64”
  region = var.droplet_region
  size = var.droplet_size
}
```

Sourcing Input Values

- Terraform will prompt for variable values when executing a `plan` or an `apply`
  - If no default value is specified
- Part of the command line option when executing a `plan` or an `apply`

```
terraform plan -var='droplet_size="s-2vcpu-2gb"' -var='droplet_region="sfo1"'
```

- In a variable definition file `.tfvars`, separate from resource definition
  - Terraform will automatically use the file if it is called `terraform.tfvars`
  - Don't forget to `.tfvars` files to `.gitignore`

```
terraform plan -var-file=values.tfvars
```

Sourcing Input Values

- From environment variables prefixed with `TF_VAR_` followed by the variable's name
  - `export TF_VAR_droplet_size="s-2vcp-2gb"`

Common Functions

- Find the maximum

```
max(5, 1, 4)
```

- Convert a string to a number, the second parameter is the base

```
parseint("42", 16)
```

- Split a string with the given delimiter, returns a list

```
split("one,two,three", ",")
```

- Concatenate a list into a string with the given delimiter

```
join(",", ["one", "two", "three"])
```

- Interpolate a string with a set of given values like printf
  - Similar to HCL string interpolation

```
format("myapp-%03d", var.count)
```

- List of functions
  - https://www.terraform.io/docs/language/functions

Complex Type - List

Use the `list` keyword to define a list
The type is defined within the ( )

```
variable ubuntu_images {
  type = list(string)
  default = [ "ubuntu-18-04-x64", "ubuntu-20-04-x64",
     "ubuntu-20-10-x64"]
}

resource digitalocean_droplet ubuntu_20_04 {
  ...
  image = var.ubuntu_images[1]
}
```

Zero based index

Common List Functions

- List length

```
length(var.ubuntu_images)
```

- Concatenate 2 or more list, returns a new list

```
concat(var.ubuntu_images,
  var.debian_images)
```

- Returns the index of given value in a list

```
index(var.ubuntu_images,
  “ubuntu-20-04-x64”)
```

- Returns an element from the list

```
element(var.ubuntu_images, 2)
```

- Extract a sublist from a list from [start, end)

```
splice(var.ubuntu_images,
  index(var.ubuntu_images,
     “ubuntu-20-04-x64”),
  length(var.ubuntu_images))
```

- List membership

```
contains(var.ubuntu_images,
  “centos-8-x64”)
```

Complex Type - Map

```
variable region_image {
  type = map(string)
  default = {
    sgp1 = "ubuntu-20-04-x64"
    lon1 = "ubuntu-20-04-x64"
    nyc1 = "ubuntu-18-04-x64"
  }
}
```

Key-value pairs

Use the `map` keyword to define a list
The value is defined within the ( )
The key is type string

```
resource digitalocean_droplet droplet {
  name = "droplet"
  region = var.region
  image = var.region_image[var.region]
  ...
}
```

```
variable region {
  type = string
  validation {
    condition = contains(
        ["sgp1", "lon1", "nyc1", var.region)
    error_message = "Supported regions are sgp1, lon1, nyc1."
  }
}
```

Variable validation rules

Condition must evaluate to true

Common Map Functions

- Returns all keys as a list

```
keys(var.region_image)
```

- Returns all values as a list

```
values(var.region_image)
```

- Lookup a key in a map, returns default value if key does not exists

```
lookup(var.region_image,
   var.a_region, "sgp1")
```

- Creates a map from 2 list

```
zipmap(
  ["sgp1", "nyc1", "lon1" ],
  ["ubuntu-20-04-x64",
    "ubuntu-18-04-x64",
    "ubuntu-20-04-x64" ]
)
```

Structural Type - Object

Objects are defined with `object`. Attribute type are defined within the `object` keyword.

Attribute definition

Object instance

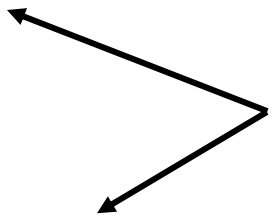
```
variable fw_ingress {
  type = object({
    protocol = string
    port_range = string
    source_addresses = list(string)
  })

  default = {
    protocol = "tcp"
    port_range = "8000-9000"
    source_addresses = [ "0.0.0.0/0", "::/0" ]
  }
}
```

```
resource digitalocean_firewall web {
  name = "web"
  droplet_ids = [ ... ]

  inbound_rule {
    protocol = var.fw_ingress.protocol
    port_range = var.fw_ingress.port_range
    source_addresses =
       var.fw_ingress.source_addresses
  }
}
```

Accessing Output Values

- Output values are attributes from provisioned resources
  - Eg. IP address, resource id, endpoint
  - Displayed after the resources are provisioned
- Output value arguments, optional except value
  - `value` - value to be bound to this output
  - `description` - describe the output
  - `sensitive` - value will be redacted if this argument is set to true

```
output ipv4 {
  value = digitalocean_droplet.ubuntu_20_04.ipv4_address
}


output monthly_price {
  value = digitalocean_droplet.ubuntu_20_04.price_monthly
  sensitive = true
}
```



<resource_name>.<name>.<attribute>

Displaying Output Values

- Output values are displayed when resources are provision
- Display the output values after resources are provisioned
  - List all the output values

    ```
    terraform output
    ```

  - List all output values in JSON forma

    ```
    terraform output -json
    ```

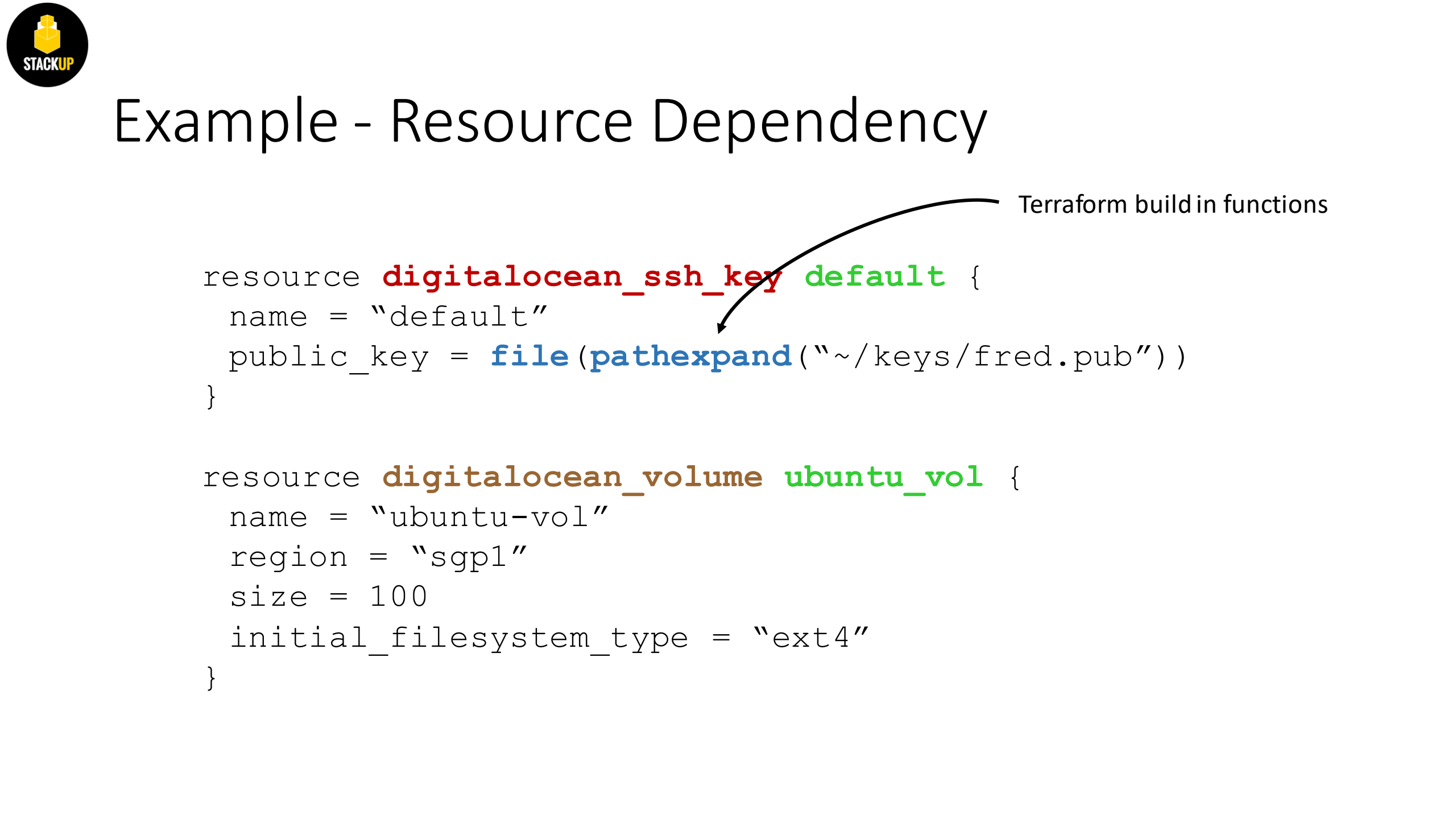
  - List a specific output value

    ```
    terraform output ipv4
    ```

Resource Dependency

- Resources are often dependent on each other
  - Eg. volumes must exist if they are to be attached to a virtual machine
  - Eg. list of VM IP addresses for configuring API gateway upstream
- Dependencies impose an order in which the resources are created
- Terraform evaluates and generate a dependency graph from the scripts
  - Serialize resource creation if resources are dependent on each other
  - Parallelize resource creation if resources are independent of each other
- Dependency graph provides safety and also reduces provisioning time

Example - Resource Dependency

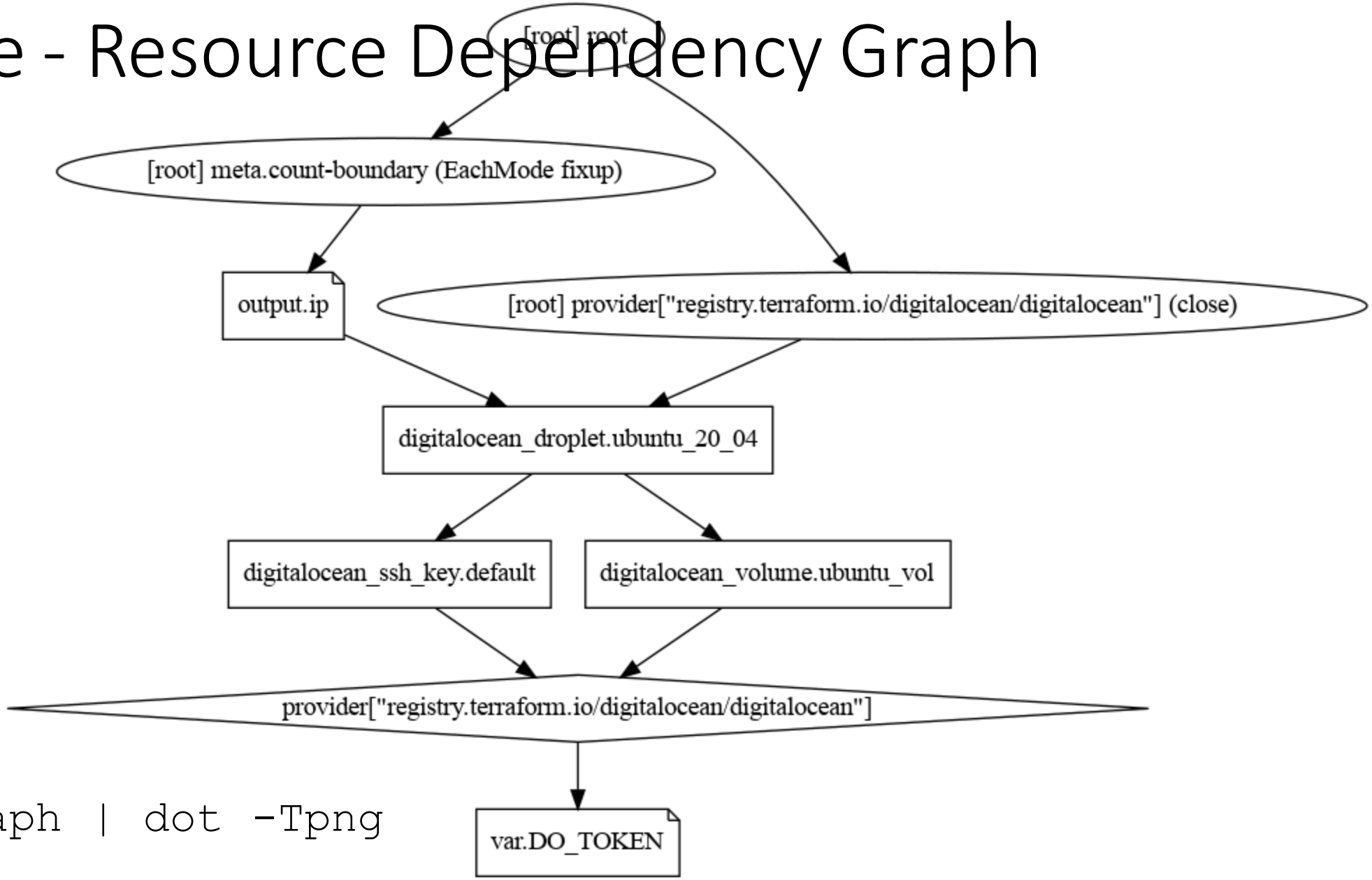



```
resource digitalocean_ssh_key default {
  name = "default"
  public_key = file(pathexpand("~/keys/fred.pub"))
}

resource digitalocean_volume ubuntu_vol {
  name = "ubuntu-vol"
  region = "sgp1"
  size = 100
  initial_filesystem_type = "ext4"
}
```

Example - Resource Dependency

```
resource digitalocean_droplet ubuntu_20_04 {
  name = "my-database"
  image = "ubuntu-20-10-x64"
  region = "sgp1"
  size = "s-1vcpu-1gb"

  ssh_keys = [ digitalocean_ssh_key.default.fingerprint ]
  volume_ids = [ digitalocean_volume.ubuntu_vol.id ]

  monitoring = true
}
```

Example - Resource Dependency Graph




```
terraform graph | dot -Tpng
```

Meta Argument - `count`

- `count` argument can be added to any resources
- Creates multiple instances of the resource
  - Eg. multiple nodes for a Kubernetes cluster
- Multiple copies of the resource is saved to a list
  - `<resource>.<name>` - the list of created resources
  - `<resource>.<name>[1]` - the 2nd instance of the resource
- `count.index` returns index (0 based) corresponding to the current iteration
  - Typically append to the name argument to create unique names for the resources

Example - `count` meta-argument

```
resource digitalocean_volume vol {
  count = var.num_of_nodes
  name = "vol-${count.index}"
  region = var.droplet_region
  size = var.volume_size
  initial_filesystem_type = "ext4"
}

resource digitalocean_droplet web {
  count = var.num_of_nodes
  name = "web-${count.index}"
  image = var.droplet_image
  region = var.droplet_region
  size = var.droplet_size
  volume_ids = [ digitalocean_volume.vol[count.index] ]
}
```

Create `num_of_nodes` instances of volume and droplet

Use `count.index` to create unique names

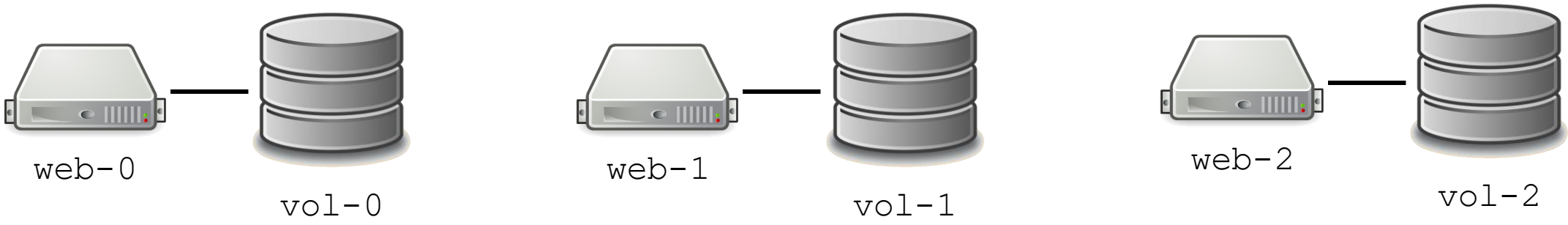
The name of the instances are `vol-0`, `vol-1`, `vol-2`, `web-0`, `web-1` and `web-2`.

Resources are stored in the two lists called `digitalocean_droplet.vol` and `digitalocean_droplet.web`

Use `count.index` to assign the volume to its corresponding droplet

Note: volumes are attached not mounted. Need other tools to mount the volume

Example - `count` meta-argument



Output of droplet_ipv4s is a list of comma separated IPv4 addresses

```
output droplet_ipv4s {
  value = join(",", digitalocean_droplet.web[*].ipv4_address)
}
```

Splat expression

Return all the `ipv4_address` attribute from all instances in the list

List Comprehension

- Splat expression is a succinct expression of the more general list comprehension
  - Similar to Python's list comprehension

Control variable, each element from the list is assigned to s

```
[ for s in a_list: s ]
```

```
output droplet_ipv4s {
  value = join(",", digitalocean_droplet.web[*].ipv4_address)
}
```

Rewritten with list comprehension

```
output droplet_ipv4s {
  value = join(",", [ for d in digitalocean_droplet.web: d.ipv4_address ])
}
```

List Comprehension

- The iterable can either be a list or a map
  - List - `for s in a_list`
  - Map - `for k, v in a_map`
- Result of a list comprehension can either be a list or an object

```
{ for d in digitalocean_droplet.web: d.name => d.ipv4_address }
```

Key

Value

- List comprehension supports filtering

Filter

```
output failed_droplet {
  value = [ for d in digitalocean_droplet.web: d.name if d.status == "failed" ]
}
```

Meta Argument - `for_each`

- Similar to `count`, but iterate over a map or set
- Create multiple copies of unique resource instead of the same resource like `count`
- `each.key` and `each.value` is the same for set

```
resource digitalocean_droplet app_server {
  for_each = var.servers
  name = each.key
  image = each.value.image
  region = each.value.region
  size = each.value.size
  ...
}
```

```
variable servers {
  type = map(
    object({
      image = string
      region = string
      size = string
    })
  )

  default = {
    database: {
      image = "g-4vcpu-16gb"
      ...
    }
    web: {...}
    proxy: { ... }
  }
}
```

List of servers
All different specs

Meta Argument - `for_each`

- `for_each` can dynamically create blocks within a resource dynamically
- `count` can only create top level resources

```
resource digitalocean_firewall web {
  name = "web"
  inbound_rule {
    protocol = "tcp"
    port_range = "22"
    source_addresses = [ "0.0.0.0/0", "::/0" ]
  }
  inbound_rule {
    protocol = "tcp"
    port_range = "8000-9000"
    source_addresses = [ "0.0.0.0/0", "::/0" ]
  }
  outbound_rule {
    protocol = "tcp"
    port_range = "53"
    destination_addresses = [ "0.0.0.0/0", "::/0" ]
  }
  outbound_rule {
    protocol = "udp"
    port_range = "53"
    destination_addresses = [ "0.0.0.0/0", "::/0" ]
  }
}
```

Repeat these blocks

Repeat these blocks

Example - Dynamic Block

```
variable ingress_rules {
  type = map(
    object({
      protocol = string
      port_range = string
      source_addresses = list(string)
    })
  )
  default = {
    "ing0": {
      protocol = "tcp"
      port_range = "22"
      source_addresses =
          ["0.0.0.0/0", "::/0" ]
    }
    ...
  }
}
```

```
resource digitalocean_firewall web {
  name = "web"

  dynamic inbound_rule {
    for_each = var.ingress_rules
    iterator = rule
    content {
      protocol = rule.value.protocol
      port_range = rule.value.port_range
      source_addresses =
          rule.value.source_addresses
    }
  }
  dynamic outbound_rule {
    ...
  }
}
```

Dynamic block to for `inbound_rule` block

Define a control variable

`inbound_rule` arguments

Assign the map to `for_each` within the dynamic

Tainting Resources

- `taint` forces Terraform to destroy and recreate a resource that it manages on the next `apply` operation
  - Eg. Rollback changes, target only specific servers instead of recreating the entire infrastructure
  - `untaint` reverses the operation
- When a resource is tainted, all resources that are dependent on it will also be recreated
  - Eg. tainting a volume will cause the server that attaches it
- Use `plan` to see the effects of a taint

```
terraform taint <resource_name>.<name>
terraform untaint <resource_name>.<name>
```

State

- Terraform keeps track of resources state in a state file
  - Created after performing an apply operation
- `terraform.tfstate`
  - JSON file
  - Maps what is defined in scripts to their actual representation
  - In structure of the file should be considered private viz. can change without notice
- Can be checked into code repository but ensure that there are no sensitive information
  - State file is in plain text

Managing State File

- List the resources in the state file

```
terraform state list
```

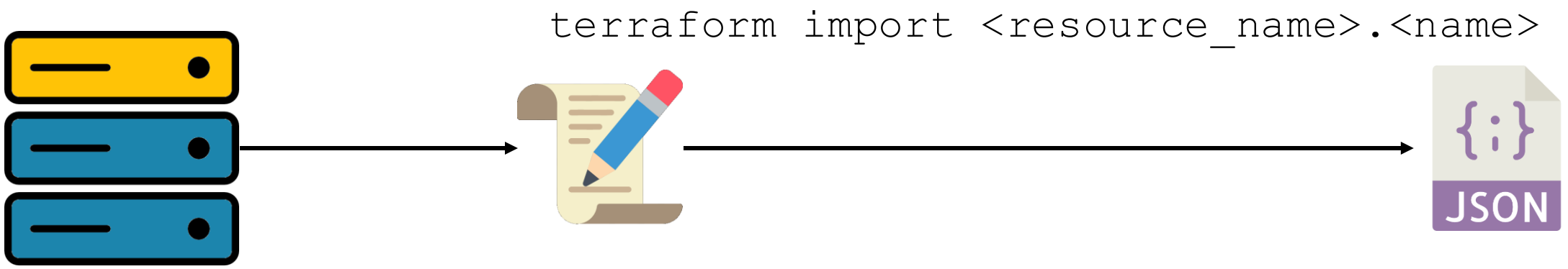
- List a specific resource instance

```
terraform state show <resource_name>.<name>
```

- Need to manually recreate resource by importing the resource if state file is destroyed
  - Eg. accidently deleted
  - Eg. manually add a provisioned resource into your state file

Importing Resources

- Manually reconcile existing resources with Terraform scripts
- Use cases
  - Accidentally deleting the state file
  - Let Terraform manage an existing resource
- Provider must support import



Existing server that needs to be managed by Terraform

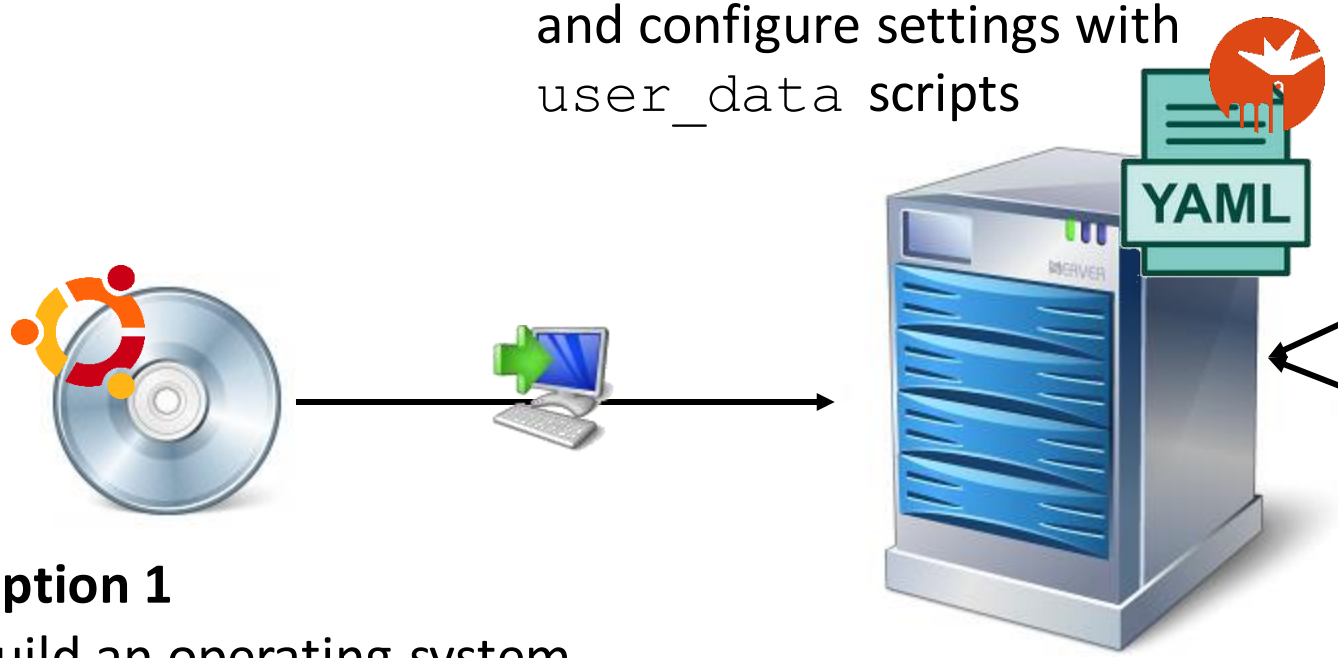
Update script to reflect the existing resource

Import the resource into the state file

Configuration Options

Option 1
Build an operating system image with additional packages and pre-configured settings

Option 2
Install additional packages and configure settings with `user_data` scripts



Option 3
Use configuration tools to install and configure system

Option 4
Use providers to install and configure system by ‘sshing’ into the system

Provisioner

- Provisioners are a set of actions that can be performed once the resource is provisioned
  - Eg. install Nginx, create users and groups, enable certain optional services
- Currently supported provisioners
  - `local-exec` - executes one or more commands on the local machine
  - `file` - copies files and directories from the local machine to the provisioned machine
  - `remote-exec` - executes one or more commands on the provisioned resource (remote)
- `file` and `remote-exec` requires a `connection` object to configure connection to the local machine to the resource
  - connection object can be share or per provisioner
  - https://www.terraform.io/docs/language/resources/provisioners/connection.html#argument-reference

Example - `local-exec` Provisioner

```
resource digitalocean_droplet web {
  name = "web"
  image = var.droplet_image
  size = var.droplet_size
  region = var.region
  ssh_keys = [ digitalocean_ssh_key.default.fingerprint ]

  provisioner local-exec {
    command = "mosquitto_pub -h broker -u ${var.username} -P
${var.password} -t status -m 'UP: ${self.ipv4_address}'"
  }
}
```

Command to execute on local machine.

Special `self` object to reference the parent

Example - `file` Provisioner

```
resource digitalocean_droplet web {
  name = "web"
  image = var.droplet_image
  size = var.droplet_size
  region = var.region
  ssh_keys = [ digitalocean_ssh_key.default.fingerprint ]
  provisioner file {
    source = "./myapp/"
    destination = "/app"
    connection {
      type = "ssh"
      user = var.username
      private_key = file("./default")
      host = self.ipv4_address
    }
  }
}
```

Copy the contents of `myapp` directory to `/app` directory on the resource

Connection configuration

Matching key pair

Example - `remote-exec` Provisioner

```
resource digitalocean_droplet web {
  name = "web"
  image = var.droplet_image
  size = var.droplet_size
  region = var.region
  ssh_keys = [ ... ]

  connection {
    type = "ssh"
    user = var.username
    private_key = file("./default")
    host = self.ipv4_address
  }
  ...
}
```

Two provisioners share a single connection block

Continue

```
...
provisioner file {
  source = "./setup.sh"
  destination = "/tmp/"
}

provisioner remote-exec {
  inline = [
    "chmod a+x /tmp/setup.sh",
    "/tmp/setup.sh"
  ]
}
```

Commands to be executed on the resource (droplet)

Provisioners are executed in the listed order

Alternative to `remote-exec`

- Options for configuring resources
  - Terraform provisioner
    - Should be last resort for configuring resources
  - Set the `user_data` (or equivalent) with a script that will be executed once the resource has been provisioned
  - Use a configuration management tool like Ansible
  - Build an image with all the required packages and configurations

Configuring with 'user_data'

- Most cloud providers provide a way to pass initialization scripts to the provisioned resource
  - The scripts will be executed on the resource once the resource is provisioned
  - https://www.terraform.io/docs/language/resources/provisioners/syntax.html#passing-data-into-virtual-machines-and-other-compute-resources
- Setup and configure server instance with cloud-init
  - Declarative way of configuring the compute resource with YAML file
  - Alternative is to use a shell script
  - See https://cloudinit.readthedocs.io/en/latest/

Example - cloud-init

```
#cloud-config

users:
- name: fred
  groups: sudo
  shell: /bin/bash
  ssh_authorized_keys:
  - ssh-rsa AAA...
package_update: true
packages:
- nginx
runcmd:
- systemctl enable nginx
- systemctl start nginx
```

This must be on the first line of the file

Create this user

Install these package(s)

Run these command

```
resource digitalocean_droplet web {
  name = "web"
  image = var.droplet_image
  region = var.region
  size = var.droplet_size
  user_data: file("./config.yaml")
}
```

Contents of this file

Templates

- Templates is used to capture provisioned information and format that information in a certain way
- Use cases
  - Write a list of provisioned server's IP address as upstreams to a Nginx configuration file
  - Generate an inventory of provisioned resources
- Templates is a form of data source
- Two ways to write template
  - In a file
  - heredoc

Creating a Template

- Any text files with
  - Variables - values are interpolated
  - Directives - control over the rendering process
  - Functions -
- Variables - `${...}`
- Directive
  - Condition - `%{if <expression>}` / `%{else}` / `%{endif}`
  - Loop - `%{for <control variable> in <collection>}` / `%{endfor}`

nginx.conf.tftpl

```
http {
  #define a block of servers
  upstream apps {
    least_conn;
    %{~ for ip in droplet_ips ~}
      server ${ip}:3000;
    %{~ endfor ~}
  }
  server {
    listen 80;
    location /app {
      proxy_pass http://apps/;
    }
  }
}
```

Rendering a Template

```
resource digitalocean_droplet app {
  count = var.instances
  ...
}

resource local_file nginx_conf {
  content = templatefile("nginx.conf.tftpl", {
    droplet_ips = digitalocean_droplet.app[*].ipv4_address
  }
  filename = "nginx.conf"
}
```

Use the `templatefile` function to render the template

Provide a map to assign values to the template variables

Appendix

Validating and Linting

- Use validate to validate the stack's files

```
terraform validate
```

- Linting is the process of checking source code for programmatic or stylistic errors
  - Linting can help enforce good practices and standardize how a program/script should be written
- `tflint` - open source linting tool for Terraform
  - https://github.com/terraform-linters/tflint

Notes

- Mounting attached volumes - DigitalOcean
  - https://blog.laurentcharignon.com/post/set-up-digital-ocean-block-storage-with-ansible/