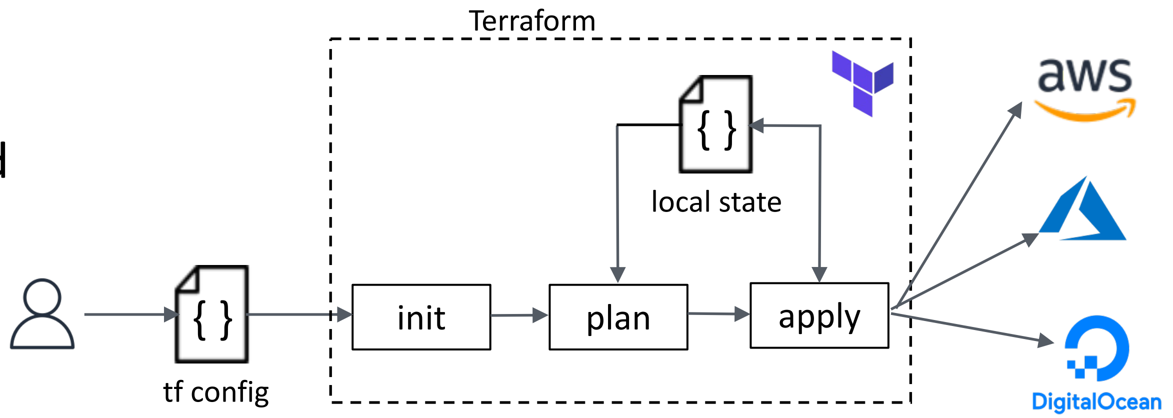
05. Managing Terraform State

Contents

Introduction

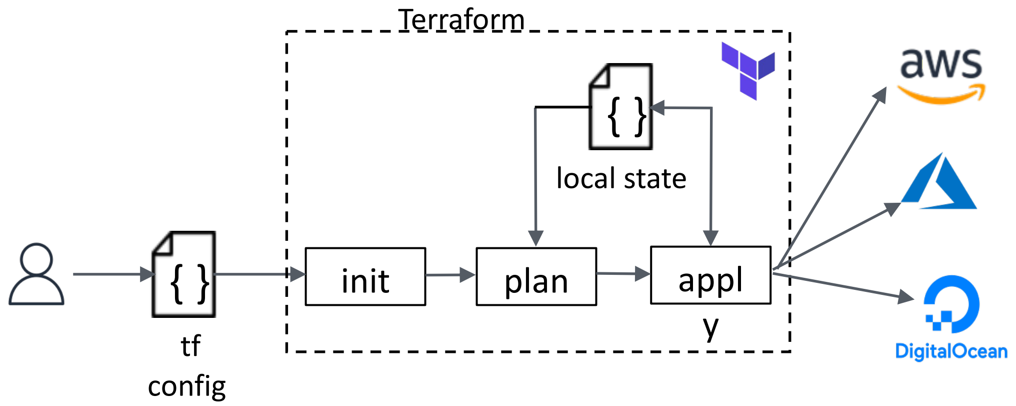
Terraform State (1)

- Terraform must store state about your managed infrastructure and configuration.
  - This state is used by Terraform to map real world resources (real world state) to your configuration (desired state), keep track of metadata, and to improve performance for large infrastructures.
  - This state is stored by default in a **local file** named "terraform.tfstate", but it can also be stored **remotely**, which works better in a team environment.
- Terraform uses the state to create plans and make changes to your infrastructure.
- Prior to any operation, Terraform does a refresh to update the state with the real infrastructure.



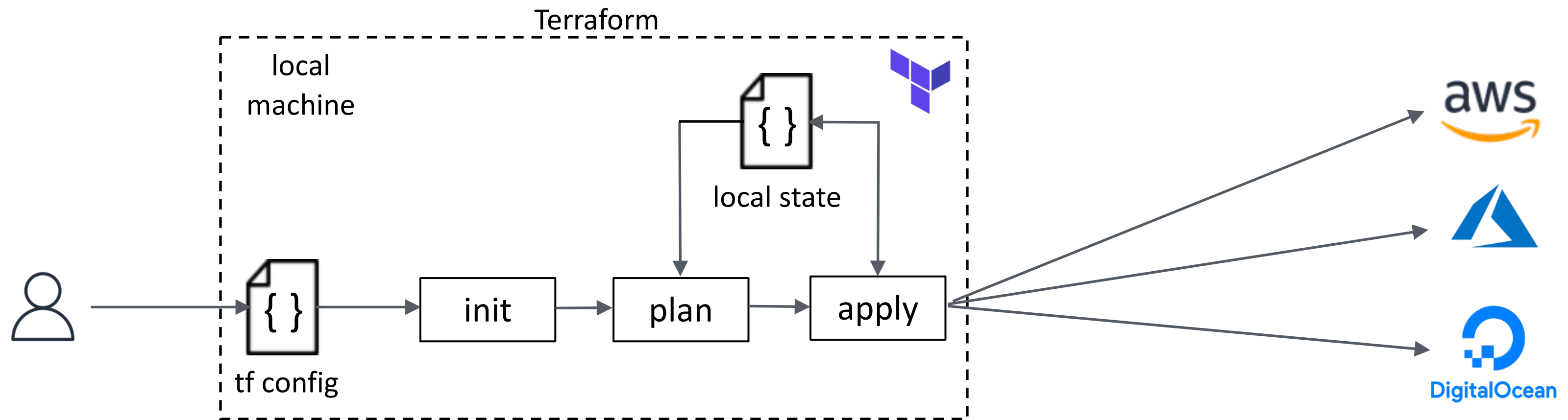
Terraform State (2)

- The primary purpose of Terraform state is to **store bindings** between objects in a remote system (e.g. actual VMs, Vnets, subnets, resource groups) and resources declared in your configuration.
  - When Terraform creates a remote object in response to a change of configuration, it will record the identity of that remote object against a particular resource instance, and then potentially update or delete that object in response to future configuration changes.
  - Terraform expects to "own" and control the remote object. Any changes to the VM, VPC, etc.. Should be performed through Terraform. Otherwise problems arise (see discusion of drift in next slides)
- Maintaining state is not trivial. Are there alternatives? : Terraform's [own explanation]



State stored locally

- By default, Terraform stores state locally in a file named *terraform.tfstate*.
- Useful in the early adoption stages of Terraform with a single user or a few users, coordinated offline.
- Does not adapt well as teams grow and more users attempt to configure the same infrastructure simultaneously



CLI Commands related to state management

- Main CLI documentation link: Manipulating Terraform State
  - Inspecting State
    - terraform state list
    - terraform state show
    - terraform refresh (alias for terraform apply –refresh-only –auto-approve)
  - Forcing Re-creation of an object (in the cloud provider)
    - terraform apply –replace <object>
    - terraform taint (deprecated) : use terraform apply –replace instead
  - Moving Resources
    - terraform state rm -terraform stops managing a specific object without destroying it
    - terraform state mv – new TF resource but maintain real object
    - See also terraform refactoring
  - Importing pre-existing infrastructure into TF state
    - Documented in the Importing Infrastructure section - need to write the TF config before hand
    - More on drift below
  - Disaster Recovery
    - terraform force-unlock (rarely used)
    - terraform state pull / push   (rarely used)

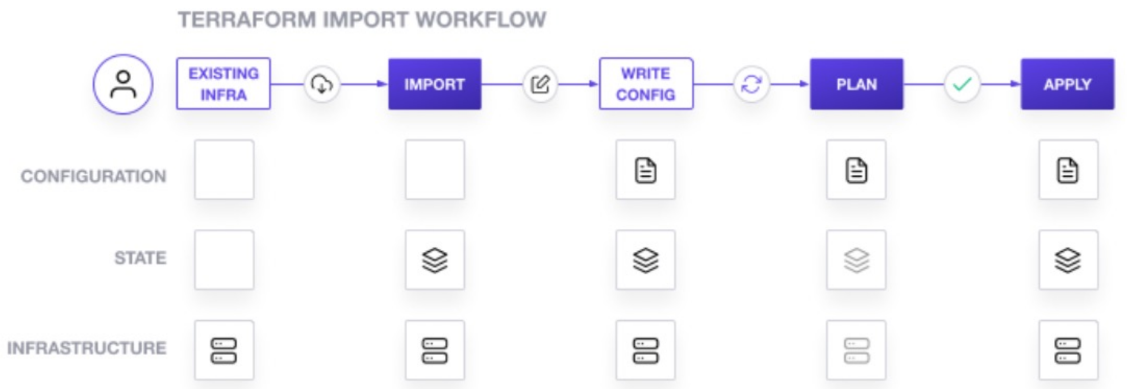
- Terraform warning about many of these commands
- **Important: Modifying state data outside a normal plan or apply** can cause Terraform to lose track of managed resources, which might waste money, annoy your colleagues, or even compromise the security of your operations.
- Make sure to keep backups of your state data when modifying state out-of-band.

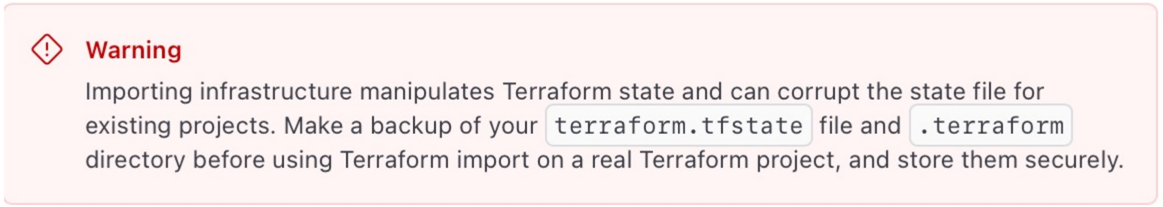
Importing Infrastructure

Lab_08_work_with_state/lab_08a_import_state

- Docs
- Workflow:

1. Identify the existing infrastructure you will import.
2. Import infrastructure into your Terraform state file. (Note: this requires at least a minimal config)
3. Write Terraform configuration that matches that infrastructure.
4. Review the Terraform plan to ensure the configuration matches the expected state and infrastructure.
5. Apply the configuration to update your Terraform state.

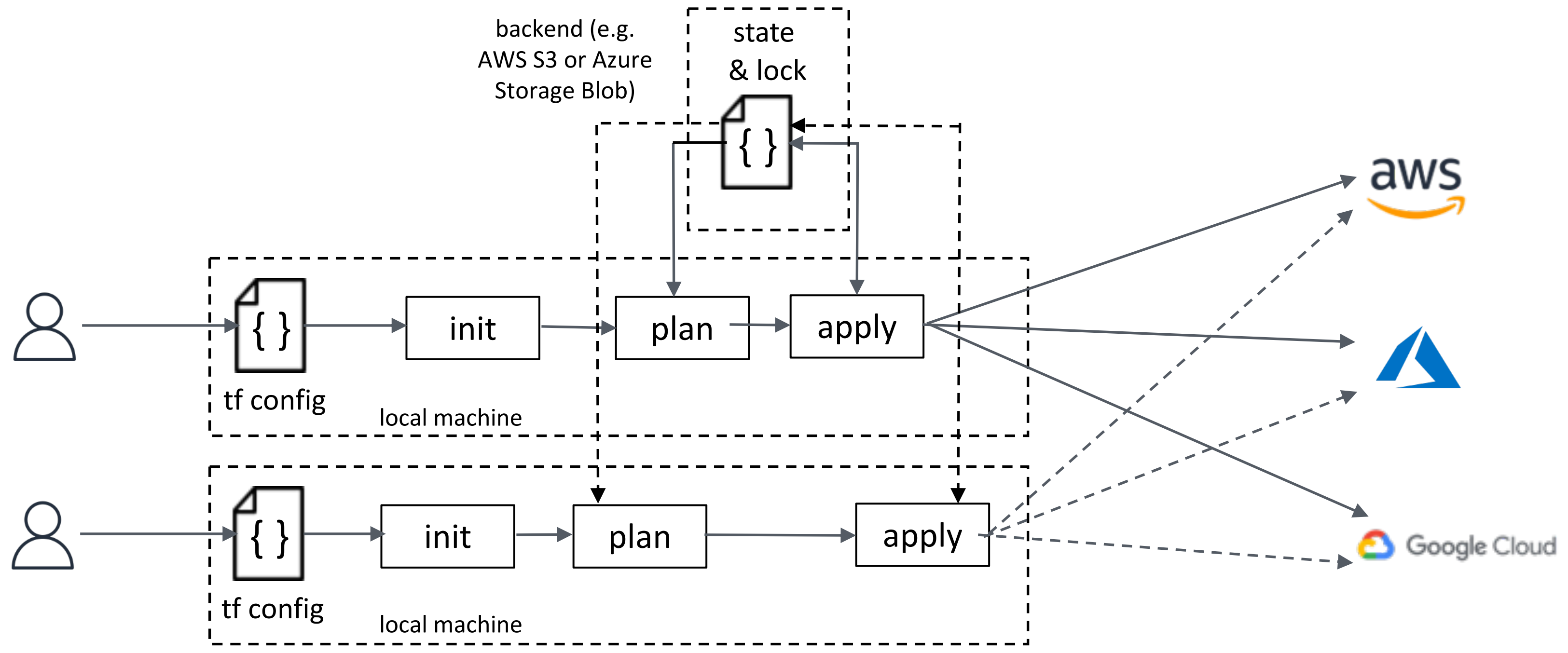



Warning

Importing infrastructure manipulates Terraform state and can corrupt the state file for existing projects. Make a backup of your `terraform.tfstate` file and `.terraform` directory before using Terraform import on a real Terraform project, and store them securely.

Remote State

- Multiple Users – Team collaboration
  - **Focus**: coordinated access to common/centralized ("remote") state in a backend
- Multiple Teams –
  - **Additional Focus**: Reusability of infrastructure code and delegation to "non-ops" users (more on this in Terraform Cloud)



State Backends

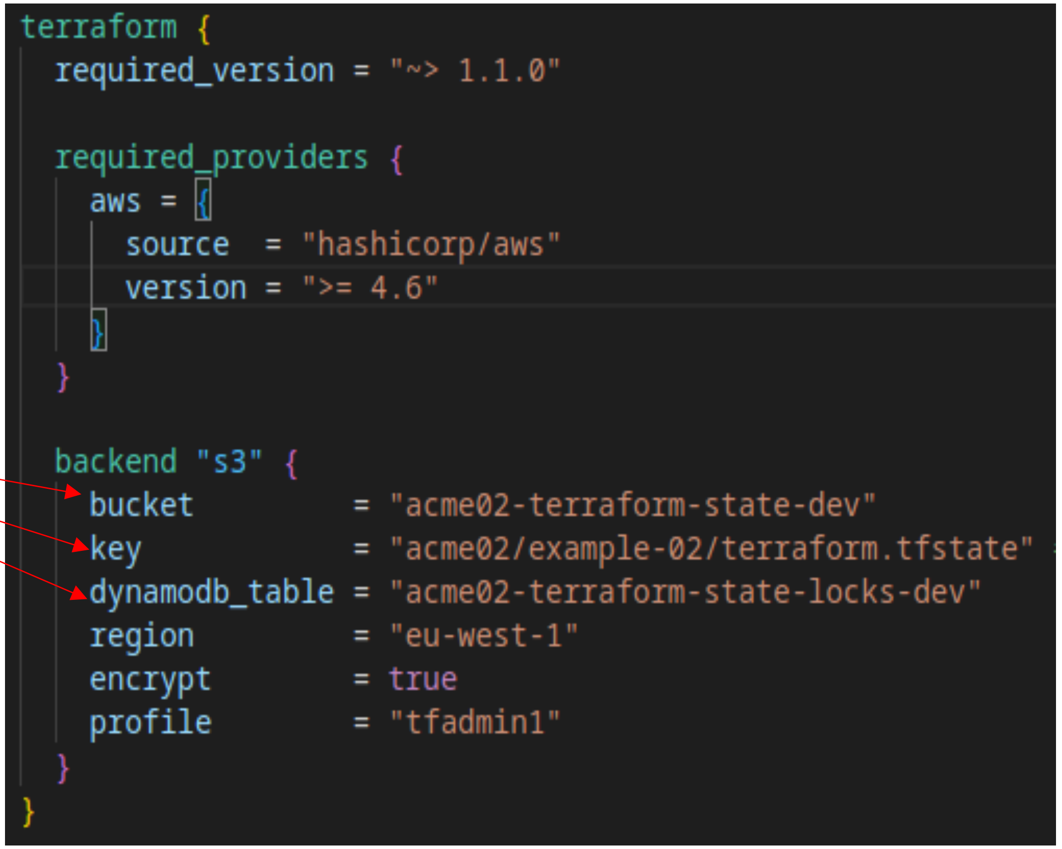
- Main TF Documentation on Backends
- Purpose of Backends
  - Store State
  - Use locks to manage access to state by multiple users
- Multiple backends supported by Terraform with different characteristics:
  - AWS s3, azurerm, consul, cos, gcs, http, kubernetes, oss, pg, etc.
  - Note several backends deprecated in release 1.2.3 (June 2022): artifactory, etcd, etcdv3, swift, manta
- HashiCorp (not surprisingly) recommends the use of the remote backend with Terraform Cloud/Enterprise

Lab_08_work_with_state/state-s3/create-state-bucket

lab_08_work_with_state/state-s3/acme02/ (use-state examples 01 & 02)

S3 Backend

- Terraform S3 Backend Documentation
- Stores state in an S3 bucket (versioning highly recommended), and implements locking with a DynamoDB table
- Big drawback – the "backend" block does not allow variables or references (e.g. to specify bucket, key and dynamodb table)
- As a consequence, backend info must be written (or copied and pasted) , modifying the app-specific key, which can be error prone and cause state overwrites between Apps.   See next slide for ways to mitigate this.
- Terraform strongly recommends enabling versioning in the bucket, to protect against accidental deletions.
- Demo / exercises with "example-01" and "example-02"

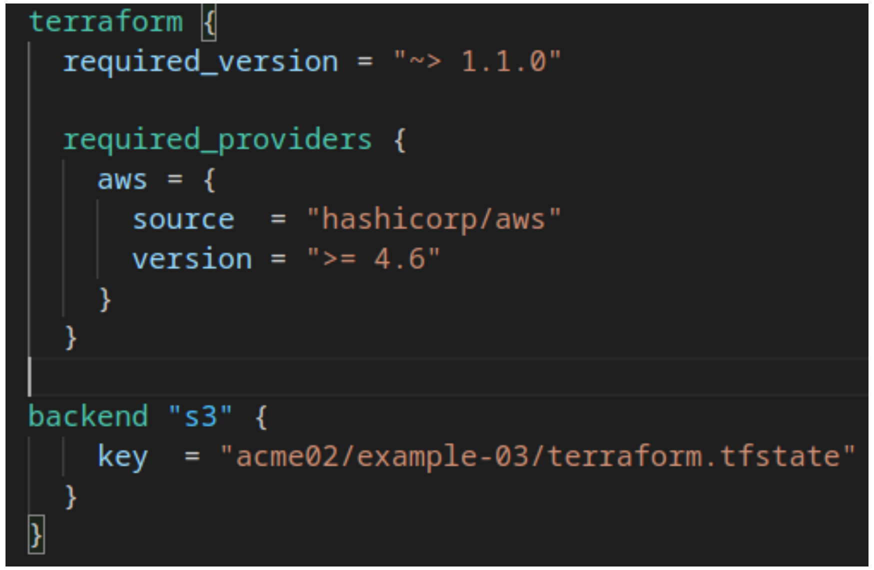


```
terraform {
  required_version = "~> 1.1.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 4.6"
    }
  }

  backend "s3" {
    bucket         = "acme02-terraform-state-dev"
    key            = "acme02/example-02/terraform.tfstate"
    dynamodb_table = "acme02-terraform-state-locks-dev"
    region         = "eu-west-1"
    encrypt        = true
    profile        = "tfadmin1"
  }
}
```

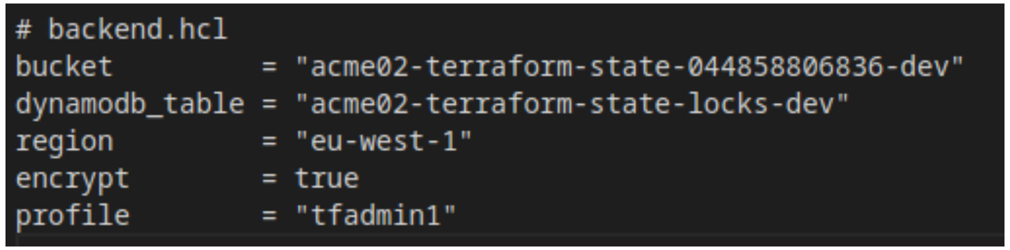
Lab_07_work_with_state/state-s3/acme02/
(use-state example 03)

S3 backend recommendation

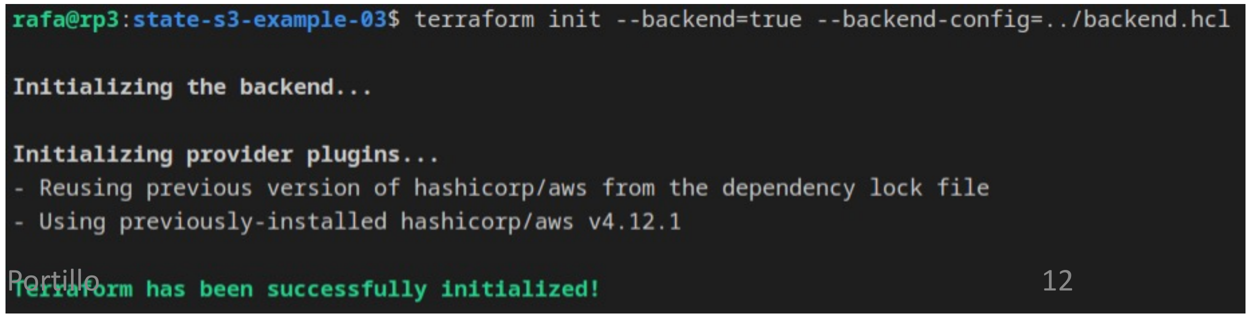
- To partially mitigate the issue discussed in the previous slide
  - Use an "quasi-empty" backend block including only the app-specific key within the bucket
  - Keep the rest of the common info info in a common backend.hcl file (e.g. per project). In this example, backend.hcl is kept in a common parent directory (../backend.hcl)
  - Run terraform init –backend-config = /path/to/backend.hcl
- Needs to be done only first time (init)
- Potential for overwriting key still exists...



```
terraform {
  required_version = "~> 1.1.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 4.6"
    }
  }

backend "s3" {
    key   = "acme02/example-03/terraform.tfstate"
  }
}
```



```
# backend.hcl
bucket          = "acme02-terraform-state-044858806836-dev"
dynamodb_table = "acme02-terraform-state-locks-dev"
region          = "eu-west-1"
encrypt         = true
profile         = "tfadmin1"
```

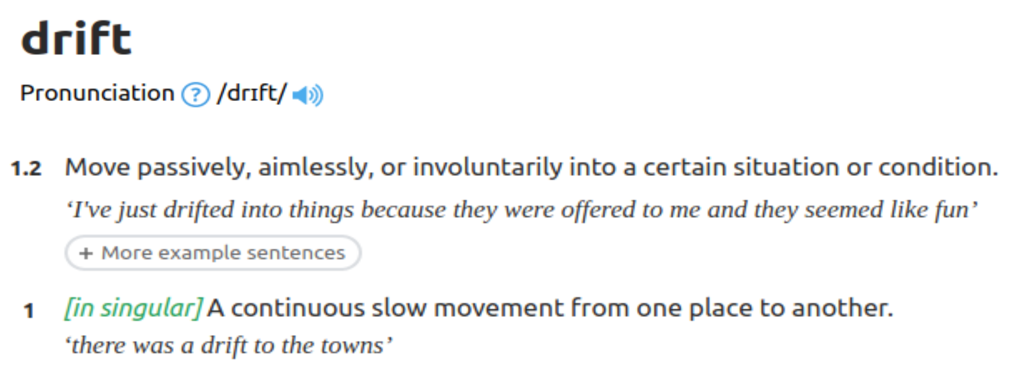


```
rafa@rp3:state-s3-example-03$ terraform init --backend=true --backend-config=../backend.hcl

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/aws v4.12.1

Terraform has been successfully initialized!
```

Drift

Drift – Definition(s)

- Drift: when the real-world state of your infrastructure differs from the state defined in your configuration



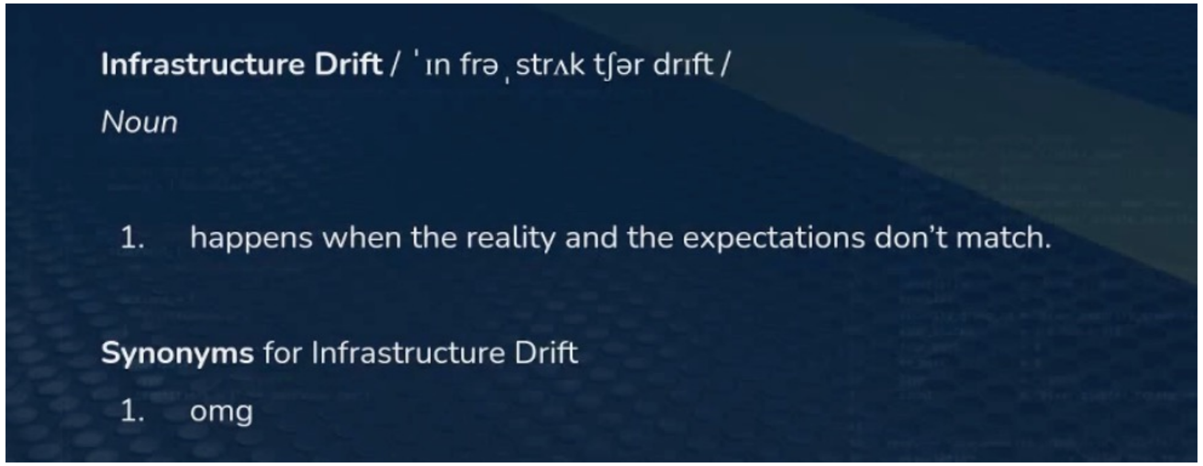
drift

Pronunciation /drɪft/

1.2 Move passively, aimlessly, or involuntarily into a certain situation or condition.
'I've just drifted into things because they were offered to me and they seemed like fun'

+ More example sentences

1 *[in singular]* A continuous slow movement from one place to another.
'there was a drift to the towns'



Infrastructure Drift / ˈɪn frəˌstrʌk tʃər drɪft /

Noun

1. happens when the reality and the expectations don't match.

Synonyms for Infrastructure Drift

1. omg

Review of terraform plan

- Docs
- **terraform plan –refresh-only**
  - creates a plan whose goal is only to update the Terraform state and any root module output values to match changes made to remote objects outside of Terraform.
  - This can be useful if you've intentionally changed one or more remote objects outside of the usual workflow (e.g. while responding to an incident) and you now need to reconcile Terraform's records with those changes.
- To actually update the state file, run **terraform apply –refresh-only** to actually update the state file
- terraform –refresh is deprecated

State and Drift - Some resources

- TF Docs Blog : Detecting and Managing Drift with Terraform (older)
- Hashicorp Tutorial on Drift (newer)
  - AWS specific: create VM in TF, add additional SG outside TF, import SG into TF
- Tool to detect drift: driftctl : (GitHub)
  - 2021 HashiTalks Presentation / Blog - The State of Infrastructure Drift: What we learned from 100+ DevOps Teams
    - By CloudSkiff, authors of the driftcl tool (CloudSkiff acquired by snyk)
    - Slides
    - Video
    - More detailed doc : State of the Infrastructure Drift 2021
- "Reverse Terraform" - Tools to generate HCL/JSON and state from existing infrastructure
  - Terrafy – MS Azure specific - see dedicated slides
  - Terraformer - active and supporting multiple clouds (not too useful for Azure as of March 2021, per "Terraform Tuesdays")
  - Terraforming (appears inactive since ~2019)

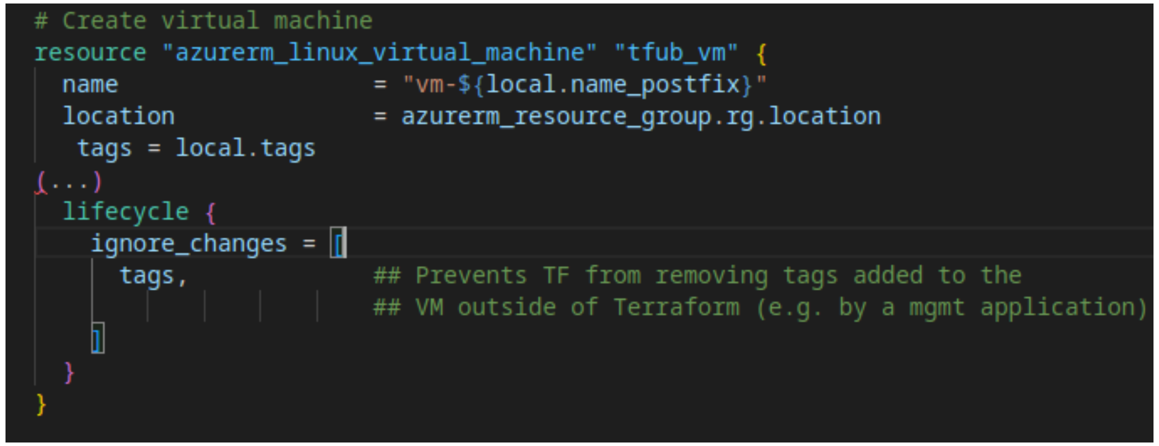
driftctl

- “driftctl is CLI tool that measures infrastructure as code coverage, and tracks infrastructure drift.”
- Open source, written in Go
- Docs and Installation

Some Useful meta-arguments related to Terraform state

Meta-argument: ignore_changes

- ignore_changes documentation
- Sometimes, non-terraform apps modify remote objects (e.g. add tags).
- Terraform will attempt to "fix" those changes, typically by deleting them if they are not included in the TF files (Terraform's declarative view of the world)
- (docs)*"The arguments corresponding to the given attribute names are considered when planning a create operation, but are ignored when planning an update. "*

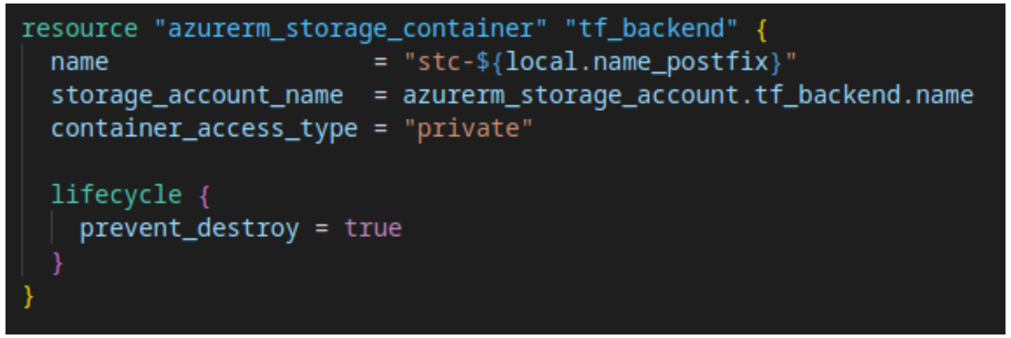


```
# Create virtual machine
resource "azurerm_linux_virtual_machine" "tfub_vm" {
  name                    = "vm-${local.name_postfix}"
  location                = azurerm_resource_group.rg.location
   tags = local.tags
(...)
  lifecycle {
    ignore_changes = [
      tags,                 ## Prevents TF from removing tags added to the
                            ## VM outside of Terraform (e.g. by a mgmt application)
    ]
  }
}
```

- (docs) *Instead of a list, the special keyword **all** may be used to instruct Terraform to ignore all attributes, which means that Terraform can create and destroy the remote object but will never propose updates to it.*

Meta-argument: prevent_destroy

- prevent_destroy documentation:
  - *This meta-argument, when set to true, will cause Terraform to reject with an error any plan that would destroy the infrastructure object associated with the resource, as long as the argument remains present in the configuration.*
  - *This can be used as a* ***measure of safety*** *against the accidental replacement of objects that may be costly to reproduce, such as database instances. However, it will make certain configuration changes impossible to apply, and will prevent the use of the terraform destroy command once such objects are created,* ***and so this option should be used sparingly.***
  - *Since this argument must be present in configuration for the protection to apply, note that this setting does not prevent the remote object from being destroyed if the resource block were removed from configuration entirely: in that case, the prevent_destroy setting is removed along with it, and so Terraform will allow the destroy operation to succeed.*



```
resource "azurerm_storage_container" "tf_backend" {
  name                  = "stc-${local.name_postfix}"
  storage_account_name  = azurerm_storage_account.tf_backend.name
  container_access_type = "private"

  lifecycle {
    prevent_destroy = true
  }
}
```