ITI8801 · Cloud Architectures and DevOps

Infrastructure as Code: Terraform

Siim Rebane · TalTech · Autumn 2026

Today

1. Why manual builds are not enough
2. What a cloud build actually is: API calls
3. Infrastructure as Code: desired state, declarative, idempotence, the tools
4. A first configuration: init, plan, apply, change, destroy
5. The file, part by part: providers, resources, references, count, variables, outputs, data sources, templates
6. State: what it is, why it exists, what is in it, where it lives
7. The commands, and reading a plan
8. This week's configuration: A2 as code, managed database, NAT gateway, bastion
9. Access control: identities, permissions, scopes
10. Limits of Terraform. Assignment 3

Why manual builds are not enough

Last week took about 90 minutes of portal work: a VNet and a subnet, two VMs, a network security group with two rules, Docker installed over SSH on both, a load balancer with a health probe, then the temporary access removed. Then you deleted it all.

What a manual build cannot give you:

- **Repeatability**: rebuilding it identically is another 90 minutes, and something will differ
- **A shared result**: a teammate's build differs from yours in ways nobody can list
- **A record**: nobody can say exactly what was built without opening every portal page
- **Review**: a colleague cannot check a build before it happens, only inspect it afterwards

What happened when you clicked

Every portal action was an HTTP request to the cloud's **management API**. Creating the VNet was, underneath:

```
PUT https://management.azure.com/subscriptions/<id>/resourceGroups/iti8801-w2
    /providers/Microsoft.Network/virtualNetworks/iti8801-vnet?api-version=2024-05-01

{ "location": "westeurope",
  "properties": { "addressSpace": { "addressPrefixes": ["10.60.0.0/16"] } } }
```

- The portal is **one client** of that API. Azure's command-line program, `az` (the Azure CLI), is another: the same requests, typed as commands. Terraform is a third
- The fields you filled in are the fields of the request. Nothing else happened
- Everything the portal can do, a program can do: the API is the cloud; the portal is a window on it

Last week's build as API calls

#	Call	Needs from earlier calls
1	create resource group	
2	create VNet	the group
3	create subnet `app`	the VNet
4, 5	create two public IPs, the temporary ones	the group
6, 7	create two network interfaces	the subnet, the public IPs
8, 9	create two VMs	the interfaces
10	create the NSG, with its two rules	the group
11	associate the NSG with the subnet	the subnet, the NSG
12, 13	create the load balancer's public IP, the load balancer with pool, probe and rule	the IP
14	add the interfaces to the backend pool	the interfaces, the pool

Fourteen calls to build, three more to lock down. Every call after the first needs an **id** returned by an earlier one: the ids force the order.

A script is not enough

The obvious next step is a script of fourteen commands:

```
az group create --name iti8801-w2 --location westeurope
az network vnet create --name iti8801-vnet --address-prefixes 10.60.0.0/16 ...
az vm create --name app-1 --subnet app ...
```

Run it and it builds. Then:

- run it **again**: `already exists` on line 1, or a second copy of everything
- **change** one thing: the script says how to create, not how to change. A second script
- **delete** everything: a third script, in reverse order, and it must know what exists
- what exists **now**? The script does not know. It ran once, some time ago

A script describes **actions**. What you want to describe is the **result**.

The idea: desired state

Write **what should exist** in files. A tool reads the files, compares them to **what does exist**, and makes the difference disappear.

```
main.tf              ⟶  terraform apply  ⟶  your cloud account
(desired state)         (compute the          (actual state)
                         difference, act)
```

Consequences of the description being a file:

- The files are **the documentation**, and cannot go stale: they are the input
- The files go in **git**: reviewed before apply, diffed, reverted
- Rebuilding is **rerunning**: 90 minutes of clicking becomes one command
- Deleting is the same tool: it knows what it created

Declarative, not imperative

Imperative (a script): the steps

```
az network vnet create --name vnet \
    --address-prefixes 10.60.0.0/16
if ! az network vnet subnet show --name app \
        --vnet-name vnet >/dev/null; then
  az network vnet subnet create --name app \
      --vnet-name vnet --address-prefixes 10.60.1.0/24
fi
```

You write the actions and the order, and every case: first run, second run, after a change.

Declarative (Terraform): the result

```
resource "azurerm_virtual_network" "vnet" {
  address_space = ["10.60.0.0/16"]
}
resource "azurerm_subnet" "app" {
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefixes     = ["10.60.1.0/24"]
}
```

You write what exists. The tool derives the actions and the order, for every case.

Reconciliation and idempotence

The tool's job is always the same: **desired** minus **actual** equals the actions.

Situation	Desired	Actual	Actions
first run	14 things	nothing	14 creates
run again	14 things	14 things	**nothing**
change a rule's port in the file	14 things	14, one differs	1 update
delete the load balancer block	13 things	14 things	1 delete
someone deleted a VM in the portal	14 things	13 things	1 create

Running the tool twice does nothing the second time. That property is **idempotence**, and it is what a script lacks: the same command is safe at any time, in any state.

Tools that do this

- **Terraform**: the most widely used, supports every major cloud. What we use
- **OpenTofu**: its open-source fork (result of a 2023 license dispute); command-compatible
- **CloudFormation / Bicep**: AWS's and Azure's own; one cloud each
- **Pulumi**: the same idea, but real programming languages instead of config files
- **Ansible**: a different job (it configures *machines*, not clouds); often used together with Terraform

The concepts in this lecture are the same in all of them; the syntax differs.

Terraform and OpenTofu

- Terraform was open source (MPL 2.0) from 2014. In August 2023 HashiCorp moved it to the **Business Source License**: free to use, but not inside a product that competes with HashiCorp's own
- The community forked the last open-source version as **OpenTofu**, under the Linux Foundation, still MPL. General availability January 2024. HashiCorp itself was bought by IBM in 2025
- Same language, same `.tf` files, same providers, same state format, same four commands: `tofu init`, `tofu plan`, `tofu apply`, `tofu destroy`
- Small differences have appeared since: OpenTofu added state encryption and a few functions; Terraform added features of its own. Nothing in this course touches either side

For this course the two are the same tool. The reference configuration runs unchanged on both; which binary you install is your choice, and the assignment does not ask.

A first configuration

One file, `main.tf`, in an empty directory: week 2's first phase, the resource group, the VNet and the subnet. Read it as a description, not a program: three things exist, each with a name and a few values.

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 4.0"
    }
  }
}

provider "azurerm" {
  features {}
  subscription_id = "5f1d33b3-..."
}

resource "azurerm_resource_group" "rg" {
  name     = "abc123-rg"
  location = "swedencentral"
}
```

```
resource "azurerm_virtual_network" "vnet" {
  name                = "abc123-vnet"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  address_space       = ["10.60.0.0/16"]
}

resource "azurerm_subnet" "app" {
  name                 = "app"
  resource_group_name  = azurerm_resource_group.rg.name
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefixes     = ["10.60.1.0/24"]
}
```

First, what running it does. Then every line of it.

`terraform init`: get the provider

```
$ terraform init
Initializing provider plugins...
- Finding hashicorp/azurerm versions matching "~> 4.0"...
- Installing hashicorp/azurerm v4.81.0...
- Installed hashicorp/azurerm v4.81.0 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above.

Terraform has been successfully initialized!
```

- Terraform itself knows nothing about Azure. The file asks for the `azurerm` plugin, and `init` downloads it into `.terraform/`: about 200 MB, not for git
- `.terraform.lock.hcl` records the exact version chosen, so that a teammate's `init` picks the same one. That file **goes into git**
- Nothing in the cloud was touched. Once per directory; again only after the list of providers changes

`terraform plan`: what would happen

```
$ terraform plan
  # azurerm_resource_group.rg will be created
  + resource "azurerm_resource_group" "rg" {
      + id       = (known after apply)
      + location = "swedencentral"
      + name     = "abc123-rg"
    }
  # azurerm_virtual_network.vnet will be created
  + resource "azurerm_virtual_network" "vnet" {
      + address_space       = ["10.60.0.0/16"]
      + id                  = (known after apply)
      + name                = "abc123-vnet"
      + resource_group_name = "abc123-rg"
      + subnet              = (known after apply)
    }
  # azurerm_subnet.app will be created
  + resource "azurerm_subnet" "app" { ... }

Plan: 3 to add, 0 to change, 0 to destroy.
```

- `+` is create. Three blocks in the file, nothing in the cloud: three creates. `plan` touched nothing; it compared
- `(known after apply)`: a value the cloud assigns at creation, the id for one. Values you never wrote appear too, `subnet` and defaults cut here: the provider lists every argument the API has

`terraform apply`: make it happen

```
$ terraform apply
  ... the same plan ...
Plan: 3 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Enter a value: yes

azurerm_resource_group.rg: Creating...
azurerm_resource_group.rg: Creation complete after 6s [id=/subscriptions/5f1d.../resourceGroups/abc123-rg]
azurerm_virtual_network.vnet: Creating...
azurerm_virtual_network.vnet: Creation complete after 5s [id=.../virtualNetworks/abc123-vnet]
azurerm_subnet.app: Creating...
azurerm_subnet.app: Creation complete after 5s [id=.../virtualNetworks/abc123-vnet/subnets/app]

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

- `apply` is the plan, a question, and then the API calls: one PUT per resource. The group first, the VNet once the group exists, the subnet once the VNet exists. The order came from the file: the VNet block uses the group's name, the subnet block uses the VNet's name
- Each `Creation complete` line carries the **id** the cloud assigned. Terraform writes them into a new file beside `main.tf`: `terraform.tfstate`
- The portal now shows `abc123-rg` with the VNet and the subnet in it, exactly as if clicked

Run it again

Change nothing, and:

```
$ terraform apply
azurerm_resource_group.rg: Refreshing state... [id=/subscriptions/.../resourceGroups/abc123-rg]
azurerm_virtual_network.vnet: Refreshing state... [id=.../virtualNetworks/abc123-vnet]
azurerm_subnet.app: Refreshing state... [id=.../virtualNetworks/abc123-vnet/subnets/app]

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.

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

- **Refreshing**: for each of the three ids in the state file, Terraform asked the cloud what that object looks like now
- Then it compared. The file says three things with these values; the cloud has those three things with these values. Nothing to do
- This is what the script did not have: the same command, any number of times, and the result is the same three things. The second run is three reads and no writes

Change a value

Edit one line in `main.tf`: the subnet's range, `10.60.1.0/24` becomes `10.60.5.0/24`. Then:

```
$ terraform plan
  ... Refreshing state ...

  # azurerm_subnet.app will be updated in-place
  ~ resource "azurerm_subnet" "app" {
      ~ address_prefixes = [
          - "10.60.1.0/24",
          + "10.60.5.0/24",
        ]
        id               = "/subscriptions/.../virtualNetworks/abc123-vnet/subnets/app"
        name             = "app"
        # (4 unchanged attributes hidden)
    }

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

- `~` is change in place: the same object, one attribute updated. The group and the VNet are not in the plan, because nothing about them differs
- You did not say "update the subnet". You changed what the subnet should look like, and the tool worked out the action
- `apply` sends one call to the API, and the state file gets the new value

Change a value that cannot change

Rename the VNet: `name = "abc123-vnet"` becomes `"abc123-net"`. Then:

```
$ terraform plan
  # azurerm_virtual_network.vnet must be replaced
-/+ resource "azurerm_virtual_network" "vnet" {
      ~ id   = "/subscriptions/.../virtualNetworks/abc123-vnet" -> (known after apply)
      ~ name = "abc123-vnet" -> "abc123-net" # forces replacement
    }

  # azurerm_subnet.app must be replaced
-/+ resource "azurerm_subnet" "app" {
      ~ virtual_network_name = "abc123-vnet" -> "abc123-net" # forces replacement
    }

Plan: 2 to add, 0 to change, 2 to destroy.
```

- The API has no call that renames a VNet. The provider knows that, and the only way to a VNet with the new name is to **destroy the old one and create a new one**: `-/+`
- The subnet was not edited and is replaced too: it lives inside the VNet, and its block references the VNet's name. One edited line, two things destroyed
- `2 to destroy` on a rename is where you stop and read. In a VNet with VMs, the VMs go with it. This plan is not applied; the name goes back

Remove a block

Delete the whole `resource "azurerm_subnet" "app"` block from the file. Then:

```
$ terraform plan
  # azurerm_subnet.app will be destroyed
  # (because azurerm_subnet.app is not in configuration)
  - resource "azurerm_subnet" "app" {
      - address_prefixes     = ["10.60.5.0/24"] -> null
      - id                   = "/subscriptions/.../subnets/app" -> null
      - name                 = "app" -> null
      - virtual_network_name = "abc123-vnet" -> null
    }

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

- The file no longer says the subnet exists; the state says it does. Desired minus actual: one destroy
- No delete command, no delete script. Removing the description removes the thing, on the next `apply`, after a `yes`
- Terraform knew which subnet to delete because the state holds its id. Something in the account that was never in the state is not Terraform's, and is left alone

Something changed outside the files

Put the subnet block back, apply. Now delete the subnet **in the portal**, and:

```
$ terraform plan
azurerm_subnet.app: Refreshing state... [id=.../subnets/app]

Note: Objects have changed outside of Terraform
  # azurerm_subnet.app has been deleted
  - resource "azurerm_subnet" "app" { ... }

Terraform will perform the following actions:
  # azurerm_subnet.app will be created
  + resource "azurerm_subnet" "app" { ... }

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

- Refresh asked the cloud about the subnet's id and got *not found*. The file still says it exists, so the plan creates it again
- The files are the truth, and `plan` reports every way reality has departed from them. That difference has a name: **drift**
- Which is why, from this week on, the portal is for looking. A change made there is undone by the next `apply`, or the files stop being true

terraform destroy

```
$ terraform destroy
  # azurerm_subnet.app will be destroyed
  # azurerm_virtual_network.vnet will be destroyed
  # azurerm_resource_group.rg will be destroyed

Plan: 0 to add, 0 to change, 3 to destroy.
  Enter a value: yes

azurerm_subnet.app: Destroying... [id=.../subnets/app]
azurerm_subnet.app: Destruction complete after 5s
azurerm_virtual_network.vnet: Destroying... [id=.../virtualNetworks/abc123-vnet]
azurerm_virtual_network.vnet: Destruction complete after 11s
azurerm_resource_group.rg: Destroying... [id=/subscriptions/.../resourceGroups/abc123-rg]
azurerm_resource_group.rg: Destruction complete after 46s

Destroy complete! Resources: 3 destroyed.
```

- The same run with an empty desired state: everything in the state file, in **reverse** order. Subnet, then VNet, then group
- The account is empty afterwards, and the bill stops. Every lab in this course ends with this command
- It works only because the state file lists what to destroy. Without it, Terraform has nothing to delete, and the portal is the only way back

The three things Terraform works with

You have now seen all three: the file you wrote, the state file that appeared next to it, and the portal.

	The files	The state	The cloud
Holds	what **should** exist	what Terraform **built**, with each thing's cloud id	what **does** exist
Written by	you	Terraform, after every apply	Terraform's API calls, and anyone with the portal
Where	`main.tf`, in git	`terraform.tfstate`, next to it, never in git	your account

Every command is the same three steps: **refresh** the state from the cloud, **compare** the files with the refreshed state, **act** on the difference. `plan` stops after the comparison. `apply` and `destroy` act, and then write the state again.

Terraform never scans the whole account. It asks only about the ids in its state; whatever else exists there is invisible to it.

One run, step by step

`terraform apply`, from the moment you press Enter:

1. **Read** every `.tf` file in the directory: the desired state
2. **Read** `terraform.tfstate`: what it built last time, and each thing's cloud id
3. **Refresh**: for every object in the state, ask the cloud for its current attributes, by id
4. **Diff** desired against refreshed: a list of creates, changes and destroys. This is the **plan**
5. Show the plan, **ask for yes**
6. **Execute**: one API call per change, in dependency order, several at once when independent
7. **Write the state** after every finished resource, so that nothing that exists is forgotten

`terraform plan` is steps 1 to 4. `terraform destroy` is the same run with an empty desired state.

The file, part by part

The first configuration had three kinds of block. The lab's configuration has three more, nothing else:

Block	Above	Describes
`terraform { }`	once	which provider plugins the configuration needs
`provider "azurerm" { }`	once	how to reach one cloud: subscription, credentials
`resource "<type>" "<name>" { }`	three times	one thing that should exist
`variable "<name>" { }`	not yet	an input: the region, a password, a switch
`output "<name>" { }`	not yet	a result printed after apply: the load balancer's address
`data "<type>" "<name>" { }`	not yet	something that exists already, read, not created

Inside every block: arguments, `name = value`.

Providers: how Terraform talks to a cloud

Terraform itself knows nothing about Azure or AWS. A **provider** is a plugin that knows one vendor's API: which resource types exist, which arguments each takes, which HTTP call creates, reads, updates and deletes one.

```
terraform {
  required_providers {
    azurerm = { source = "hashicorp/azurerm", version = "~> 4.0" }
  }
}
provider "azurerm" { features {} }
```

- `azurerm_*` resources come from the azurerm provider, `aws_*` from aws; thousands of others exist (GitHub, Cloudflare, Grafana, Docker)
- Credentials come from your logged-in CLI (`az login` / `aws login`), **never** from the files
- `~> 4.0` means any 4.x, not 5: resource definitions change between major versions

HCL: Terraform's file format

HCL (HashiCorp Configuration Language) is what `.tf` files are written in. It describes things; it does not run things. Four shapes cover nearly every line:

```
name           = "iti8801-vnet"                  # a string
address_space  = ["10.60.0.0/16"]                # a list
tags           = { course = "iti8801" }          # a map
location       = azurerm_resource_group.rg.location   # a reference to another block
```

A file is a set of **blocks**, `type "label" { arguments }`. Terraform reads every `.tf` file in the directory as one configuration; splitting into files is for humans.

What a resource looks like

```
resource "azurerm_virtual_network" "vnet" {
  name                = "iti8801-vnet"
  resource_group_name = azurerm_resource_group.rg.name
  address_space       = ["10.60.0.0/16"]
  location            = "westeurope"
}
```

- `resource "<type>" "<local name>"` : the type comes from the provider, the local name is yours and appears nowhere in the cloud
- The arguments mirror what the portal asked you, so you already know what they mean
- This one block is the PUT request the portal sent in week 2. The provider builds the request from the arguments

Arguments and attributes

A resource has two kinds of values:

- **Arguments**: what you write. `name`, `address_space`, `location`. Inputs to the API call
- **Attributes**: what the cloud returns after creation. `id`, a VM's `private_ip_address`, a database's `fqdn`. Outputs of the API call, stored in the state

Both are readable from other blocks with the same syntax, `type.name.attribute`:

```
azurerm_virtual_network.vnet.name          # an argument you wrote
azurerm_virtual_network.vnet.id            # an attribute the cloud assigned
azurerm_linux_virtual_machine.app.private_ip_address
```

Before the resource exists, an attribute has no value. The plan prints it as `(known after apply)`.

References: how values flow between resources

Nothing in a real setup stands alone: the subnet needs the VNet's name, the network interface needs the subnet's id, the VM needs the interface's id. In HCL that is a **reference**:

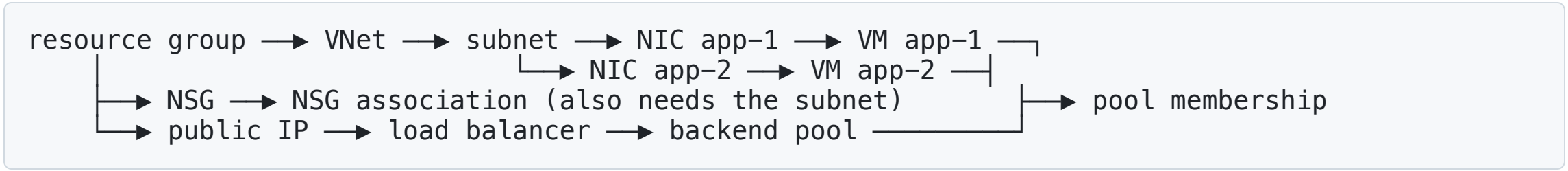
```
resource "azurerm_subnet" "app" {
  virtual_network_name = azurerm_virtual_network.vnet.name   # ← from the VNet block
  address_prefixes     = ["10.60.1.0/24"]
}

resource "azurerm_network_interface" "app" {
  ip_configuration { subnet_id = azurerm_subnet.app.id }    # ← from the subnet block
}
```

A reference does two things at once: it carries the **value**, and it tells Terraform the **order**. The subnet cannot be created before the VNet's name exists, so the VNet goes first.

The dependency graph

Terraform collects every reference into a graph and walks it. Last week's build:



- Everything with nothing between it is created **at the same time**: both interfaces, the NSG, the public IP, in parallel
- Destroy walks the same graph **backwards**: VMs before interfaces, interfaces before the subnet, the subnet before the VNet
- You never write any of this. The graph is the references

Explicit dependencies

Sometimes the order matters but no value flows. Terraform cannot see that, so you say it:

```
resource "azurerm_postgresql_flexible_server" "db" {
  delegated_subnet_id = azurerm_subnet.db.id
  private_dns_zone_id = azurerm_private_dns_zone.db.id
  depends_on          = [azurerm_private_dns_zone_virtual_network_link.db]
}
```

- The server references the zone, so the zone is created first. But the zone's **link** to the VNet is a separate resource that the server never references, and the server fails to create if the link does not exist yet
- `depends_on` adds the missing edge to the graph
- Rare, and a sign of an ordering rule the API has and the provider does not express. This week's configuration has exactly this one

count: several from one block

Last week had two identical app VMs. One block, with `count`:

```
resource "azurerm_linux_virtual_machine" "app" {
  count = 2
  name  = "${var.prefix}-app-${count.index + 1}"       # app-1, app-2
  network_interface_ids = [azurerm_network_interface.app[count.index].id]
}
```

- `count.index` is 0 and 1: the only thing that differs between the copies
- The resources are `azurerm_linux_virtual_machine.app[0]` and `app[1]`; the plan prints them that way
- All of them at once: `azurerm_linux_virtual_machine.app[*].private_ip_address` is a list

A counted resource is referenced **with its index**, always. `app.id` does not exist; `app[0].id` does.

count as a switch

`count` may be 0. Then the block describes nothing, and the resource does not exist:

```
variable "locked" { type = bool, default = false }

resource "azurerm_nat_gateway" "nat" {
  count = var.locked ? 0 : 1
  ...
}

resource "azurerm_subnet_nat_gateway_association" "app" {
  count          = var.locked ? 0 : 1
  nat_gateway_id = azurerm_nat_gateway.nat[0].id       # index 0, because count
}
```

- `condition ? a : b` is an ordinary conditional expression
- Changing `locked` from false to true makes the plan say **1 to destroy** for every block with that count. The same files describe both phases of last week's lab

Variables: what the file leaves open

The first configuration had the subscription and the region written into it. Some values are not the configuration's to fix: your subscription, your region, your password, your SSH key, your home address for the bastion rule. And one name: the database server's, which on Azure becomes a hostname on the internet and must be unique worldwide. A **variable** is a value the file leaves open:

```
variable "location"    { type = string }                          # no default: must be given
variable "vm_size"     { type = string, default = "Standard_B2ats_v2" }
variable "db_password" { type = string, sensitive = true }
```

Used as `var.location`, anywhere a value can stand: `location = var.location`.

- **Types** are checked before anything runs: a bool where a string is expected stops the plan
- **Default** makes a variable optional; no default makes Terraform ask, or fail in a script
- **sensitive** hides the value from plan output and the terminal. It is still in the state, in plain text

The same files build every student's setup. What differs is in the variables, never in the files.

Setting variables

Five ways, and if several apply, the later wins:

Way	Example	Use
default in the block	`default = "westeurope"`	the sensible value
environment variable	`export TF_VAR_prefix=abc123`	scripts, pipelines
`terraform.tfvars`	a file of `prefix = "abc123"` lines, read automatically	your own values
`-var-file`	`terraform apply -var-file=lab.tfvars`	several sets of values
`-var`	`terraform apply -var locked=true`	a one-off change

A `*.tfvars` file with your password and prefix in it is **yours**, and never goes into git. The `.gitignore` in the reference configuration excludes it.

Outputs: what apply tells you

```
output "lb_address" { value = azurerm_public_ip.lb.ip_address }
output "db_host"    { value = azurerm_postgresql_flexible_server.db.fqdn }
```

```
$ terraform apply
...
Outputs:
lb_address = "20.31.44.10"
db_host    = "abc123-db.postgres.database.azure.com"
```

- Printed after every `apply`; `terraform output lb_address` prints one, from the state, at any time
- The way you learn the load balancer's address for the validator without opening the portal
- Also how one configuration hands values to another, and to scripts around it

Data sources: reading what already exists

A `resource` block creates something. A `data` block **reads** something that exists, created elsewhere or by the cloud itself:

```
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"]            # Canonical
  filter { name = "name", values = ["ubuntu/images/*/ubuntu-*-24.04-amd64-server-*"] }
}
```

An AMI (Amazon Machine Image) is AWS's name for a VM image. `data.aws_ami.ubuntu.id` is the current Ubuntu image, looked up at plan time. Nothing is created, nothing is destroyed, and the configuration does not hard-code an id that goes stale.

Other uses: your own public IP, an existing resource group, the list of availability zones in a region.

Templates: a file with holes in it

Last week you typed the Docker command over SSH. This week the VM runs it at boot from a file, and Terraform fills the file's holes first:

```
runcmd:                                                          # cloud-init-app.yaml
  - docker run -d --restart=always --network host -e DB_TARGET=${db_host}:5432 ${beacon_image}
```

```
custom_data = base64encode(templatefile("cloud-init-app.yaml", {
  db_host      = azurerm_postgresql_flexible_server.db.fqdn
  beacon_image = var.beacon_image
}))
```

- `templatefile(path, map)` reads the file and replaces each `${name}` with the map's value
- `base64encode` is the form the VM API wants the boot script in
- The database's hostname flows from the database block, through the template, into the VM. Nobody types it

Cloud-init: setup without SSH

cloud-init is the program that runs first on a fresh Linux VM in every cloud. It reads the boot script the cloud passes in (`custom_data` on Azure, `user_data` on AWS) and does what it says: packages to install, files to write, commands to run.

```
#cloud-config
packages: [docker.io]
runcmd:
  - docker run -d --restart=always --network host ...
```

- The VM configures **itself**, identically, every time it is created
- No SSH rule to add and forget to delete; no public IP on a VM at any point
- A machine built this way is **destroyed and recreated**, never repaired: the fix goes into the file, and the next apply builds a correct machine

Expressions: the few you will meet

Anywhere a value stands, an expression may stand:

Expression	Means
`"${var.prefix}-vnet"`	string with a value inside
`var.locked ? 0 : 1`	conditional
`azurerm_subnet.app.id`	reference
`azurerm_linux_virtual_machine.app[*].private_ip_address`	all instances of a counted resource, as a list
`cidrsubnet("10.60.0.0/16", 8, 1)`	`10.60.1.0/24`: the n-th subnet of a range
`templatefile(...)`, `base64encode(...)`, `file(...)`	functions

`terraform console` evaluates any expression against the current state. Try `cidrsubnet` there.

State: what it actually is

`terraform.tfstate` is a JSON file, written by Terraform, in the working directory:

```
{ "resources": [
    { "type": "azurerm_virtual_network", "name": "vnet",
      "instances": [ { "attributes": {
          "id": "/subscriptions/1a2b.../virtualNetworks/abc123-vnet",
          "name": "abc123-vnet",
          "address_space": ["10.60.0.0/16"],
          "location": "westeurope" } } ] },
    { "type": "azurerm_linux_virtual_machine", "name": "app", "instances": [ ... ] }
] }
```

One entry per resource instance: the block that describes it, the cloud's **id** for it, and every attribute the cloud reported last time.

Why state must exist

Your file says `resource "azurerm_virtual_network" "vnet"`. The cloud has a VNet named `abc123-vnet`. Nothing connects the two: the local name `vnet` exists only in your file, and names in the cloud are not unique across groups, subscriptions, or time.

The state is the **link**: *block* `vnet` *in the file is cloud object* `/subscriptions/.../abc123-vnet`.

Without it, on the second run Terraform has no way to know the VNet is already there:

- `plan` shows every resource as **to create**, and `apply` builds a second copy of everything, or fails on the names
- `destroy` has nothing to destroy: **the infrastructure keeps running and billing**, and the portal is the only way to remove it

The state is not a cache. It is the only record of what the files correspond to.

What state contains, and why that matters

Every attribute of every resource, as the cloud returned it:

- ids, names, addresses, sizes: harmless
- the database **password** you passed as a variable: plain text
- a VM's admin password, a storage account's access keys, anything a provider returns: plain text

`sensitive = true` hides a value from the terminal. It does not encrypt it in the state.

So: the state file is a **secret**, and is handled like one. Not in git, not in a chat, not in a Moodle upload. The `.gitignore` in the reference configuration excludes it, and the assignment checks that it stayed excluded.

State and reality: drift

Between two runs, reality can change without Terraform: someone edits a rule in the portal, a VM is deleted by hand, the cloud retires an image. That difference is **drift**, and refresh finds it:

What happened outside Terraform	Refresh sees	Plan says
rule's port changed in the portal	attribute differs from the file	`~` change back to the file's value
a VM deleted in the portal	object not found	`+` create it again
state file deleted	nothing in state	`+` everything, on top of what exists

`plan` therefore answers "does reality still match the files?", and the answer should be *No changes*. From this week on the portal is for **viewing**: every change goes through the files, or the files stop being true.

Three rules for state

- **Never edit it by hand.** It is Terraform's, and one wrong id makes Terraform destroy or orphan the wrong thing
- **Never delete it.** Terraform forgets the infrastructure exists; the infrastructure keeps billing
- **Never commit it to git.** It contains secrets in plain text

Looking at it is fine, and useful, through Terraform:

```
terraform state list                          # every resource Terraform knows
terraform state show azurerm_lb.lb            # one resource, all attributes
terraform show                                # everything, readable
```

Read-only commands; they are how you check what the state believes before you act on it.

Where state lives

Today the state is a file in your working directory, and for one person on one laptop that is enough.

It stops being enough the moment a second person or a pipeline runs `apply`:

- two laptops, two state files, two views of one account; each `apply` fights the other
- a lost laptop is a lost state: infrastructure that nobody can destroy without the portal

Terraform's answer is a **remote backend**: the state lives in a storage account or an S3 bucket, every run reads it from there and writes it back, and a **lock** is held for the duration of a run so that two applies cannot overlap.

```
terraform {
  backend "azurerm" { storage_account_name = "...", container_name = "tfstate", key = "a3.tfstate" }
}
```

The four commands

```
terraform init      # once per checkout: downloads providers, writes the lock file
terraform plan      # refresh, diff, print WHAT WOULD CHANGE; touches nothing
terraform apply     # plan, ask, execute, write state
terraform destroy   # plan the removal of everything in the state, ask, execute
```

`plan` is the most important command of the four. Read it before every `apply`; then `apply` cannot surprise you. Two more that cost nothing:

```
terraform fmt         # formats the files the standard way; run before every commit
terraform validate    # checks syntax and references without touching the cloud
```

Reading a plan

```
  # azurerm_linux_virtual_machine.app[0] will be created
  + resource "azurerm_linux_virtual_machine" "app" {
      + name       = "abc123-app-1"
      + private_ip = (known after apply)
    }

  # azurerm_nat_gateway.nat[0] will be destroyed
  - resource "azurerm_nat_gateway" "nat" { ... }

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

- `+` create, `-` destroy, `~` change in place, `-/+` **destroy and recreate**: the one to look at twice, because the old thing dies
- `(known after apply)` is a value the cloud assigns; it becomes a real value in the state afterwards
- The summary line is the sanity check: a plan that says *to destroy* when you expected to add is wrong, and you stop

Destroy and recreate: when a change is not a change

Some arguments cannot be changed on an existing object: a VM's size can, its image cannot; a subnet's name cannot; a database's region cannot. The provider knows which, and the plan shows it:

```
  # azurerm_linux_virtual_machine.app[0] must be replaced
-/+ resource "azurerm_linux_virtual_machine" "app" {
      ~ source_image_reference { ~ sku = "22_04-lts" -> "24_04-lts" }   # forces replacement
```

- The old VM is destroyed, a new one created. Its disk, its private IP, and everything installed on it are gone
- For a stateless app VM built by cloud-init, that is fine and expected
- For a database VM with data on its disk, that is the data gone. A managed database is one answer

The word `replaced` in a plan is where you slow down.

When apply fails halfway

`apply` creates resources in dependency order, several at a time. If one fails, the error is printed, the ones already created **stay**, and the state records them.

- Read the error; it names the resource and the cloud's reason: a quota, a name taken, a region without the size
- Fix the cause, run `apply` again: Terraform continues from where it stopped, because the state knows what exists
- Nothing is ever half-created without a record; the worst case is a resource that exists and is not in the state, which `plan` shows as "to add" again

A failed apply is normal. Read, fix, rerun.

What goes into git

Commit

- every `*.tf` file
- `cloud-init-*.yaml`, `user-data-*.sh`
- `.terraform.lock.hcl`
- `.gitignore`

Never commit

- `terraform.tfstate`, `terraform.tfstate.backup`
- `*.tfvars` (your prefix, your password)
- `.terraform/` (the downloaded providers)
- `*.pem`, `id_rsa`, any key

The reference configuration's `.gitignore` already lists the right side. A repository with a state file or a tfvars file in it is an A3 that does not pass.

This week's setup: A2 as code, and more

The reference configuration, one directory per cloud, is week 2's setup written down, plus three things week 2 could not have:

File	Contains
`providers.tf`	terraform and provider versions
`variables.tf`	`prefix`, `location`, `beacon_image`, `db_password`, `admin_ssh_key`, `my_ip`, `vm_size`, **`locked`**
`main.tf`	network, subnets, firewall rules, load balancer, two VMs, the database, the NAT gateway, the bastion
`cloud-init-app.yaml`	what you typed over SSH last week, done by the VM itself at boot
`outputs.tf`	the load balancer's address, the bastion's address, the database hostname

Every click you made last week is a named block in `main.tf`. Reading it is reading what you built.

The `locked` variable

Week 2's two phases, a way out for installing and then no way out, are one boolean here:

```
resource "azurerm_nat_gateway" "nat" {
  count = var.locked ? 0 : 1
  ...
}
```

```
terraform apply                       # locked=false: NAT exists, VMs pull the image
terraform apply -var locked=true      # plan: N to destroy — READ IT
```

Removing internet access becomes a reviewable list of changes with a record, instead of portal edits nobody can retrace. On AWS the same variable removes the NAT gateway and the private subnets' route through it.

What A3 adds to A2

Three things, each a reason on its own:

1. **A managed database** in a **second private subnet**, reachable only from the app subnet. The database tier of the lecture 2 design
2. **A NAT gateway** as the way out during the build, removed by `locked`. No VM ever has a public IP
3. **A bastion host**: one small VM in a public subnet, the only way to reach the private VMs over SSH, when you need to look inside

The validator checks the same five things as A2, and one more: the beacon's report that it **reaches the database** on the private network.

A database as a service

Last week's database would have been a VM with PostgreSQL installed by hand: yours to patch, back up, and resize, and yours to lose when the disk fills.

Both clouds sell PostgreSQL as a **managed service**: you ask for a server of a size and a version, and the cloud runs it. What you get:

- backups taken and kept without you; restore to a point in time
- patching of the operating system and of PostgreSQL itself
- a hostname and a port; nothing to log in to, no disk to watch
- a bigger size is a setting, not a migration

Azure: **Azure Database for PostgreSQL, Flexible Server**. "Flexible" is only the product's name: the second generation of Azure's managed PostgreSQL, the one that can live inside your VNet; the first, "Single Server", is retired. AWS: **RDS** (Relational Database Service) **for PostgreSQL**. The smallest sizes cost cents an hour on the student credit.

Why managed, and what it costs you

The trade is control for operations:

	Database on your VM	Managed database
Install, patch, upgrade	you	the cloud
Backups and restore	you, if you remember	automatic, point in time
Where it runs	a subnet you chose	a subnet you chose, with rules the service dictates
Creation time	a minute	**five to ten minutes**
Access	SSH and `psql`	`psql` over the network only
Cost	the VM	the VM plus a service margin

The creation time shapes today's lab: the database is applied **once**, in its own step, and read about while it builds.

The database on the private network: Azure

Flexible Server on a private network needs three things that no other resource this week needs:

- a subnet **delegated** to the service: `delegation { service_delegation { name = "Microsoft.DBforPostgreSQL/flexibleServers" } }`. Nothing else may live in that subnet; that is your second private subnet
- a **private DNS zone**, `privatelink.postgres.database.azure.com`, in which the server's hostname resolves to its private address
- a **link** from that zone to the VNet, so that VMs in the VNet use it

Three resources, `azurerm_subnet`, `azurerm_private_dns_zone`, `azurerm_private_dns_zone_virtual_network_link`, then `azurerm_postgresql_flexible_server` referencing all three. The server's `fqdn` attribute (fully qualified domain name, its hostname) is what the app VMs get as `db_host`.

The database on the private network: AWS

RDS asks for less, and one thing it insists on:

- a **DB subnet group**, `aws_db_subnet_group`, listing the subnets the database may live in. RDS requires subnets in **at least two availability zones**, so the second private subnet exists twice, one per zone, like week 2's app subnets
- a **security group** for the database allowing 5432 from the app instances' group, the same group-as-source rule as week 2
- `aws_db_instance` with `publicly_accessible = false`, `skip_final_snapshot = true` so that destroy does not stall on a backup

The instance's `address` attribute is `db_host` for the app instances.

The firewall rule for the database

Lecture 2 wrote the rule; this week it becomes code:

	Rule
Azure	NSG on the database subnet: allow 5432 from `10.60.1.0/24`, priority 100; **deny** everything from `VirtualNetwork`, priority 4000, because the built-in rule would otherwise let any VM in the VNet in
AWS	security group on the database: 5432 from `app-sg`; nothing else, because security groups deny by default

The beacon's report proves the rule from the inside: `db_check` succeeds from an app VM. The same connection attempt from the bastion fails, and that failure is the rule working.

The bastion as code

One small VM in a **public subnet** with its own public IP, and two firewall rules that chain:

```
laptop ──22, from your IP only──▶ bastion 10.60.0.4 ──22, from 10.60.0.0/24 only──▶ app VMs
```

- The bastion's rule: port 22 from **one address**, yours, the `my_ip` variable
- The app subnet's rule: port 22 from **the bastion's subnet only**. Nobody else, nowhere else
- The bastion runs nothing but SSH: no Docker, no application

Same shape on both clouds: `azurerm_linux_virtual_machine` with an `azurerm_public_ip`, or `aws_instance` in the public subnet with a security group of its own.

Using the bastion

```
ssh -J student@<bastion-ip> student@10.60.1.10
```

`-J` is **jump**: SSH connects to the bastion, and through it to the private address. Your key never leaves the laptop; the bastion only forwards the connection. In `~/.ssh/config`:

```
Host app-1
    HostName 10.60.1.10
    User student
    ProxyJump student@<bastion-ip>
```

Then `ssh app-1`. From inside: `docker ps`, the beacon's log, `psql` (the PostgreSQL command-line client) to the database. Every login passes through the bastion, so its log records every login.

Azure's public IP quota, used exactly

Azure for Students allows three public IPs. This week's configuration uses all three, and none on an app VM:

Public IP	Belongs to	Exists
1	the load balancer	always
2	the NAT gateway	until `locked=true`
3	the bastion	always

So nothing from A1 or A2 may remain in the subscription when A3 starts, or the third `azurerm_public_ip` is refused with a quota error in the plan's apply.

What takes time, and in which order

Terraform creates in dependency order, several resources at once. This week's configuration builds in about this sequence:

Step	Resources	Time
network	resource group, VNet, subnets, DNS zone, firewalls	under a minute
public addresses, NAT gateway		a minute
VMs and the bastion, load balancer		two to three minutes
the database	one resource	**five minutes** (Azure), up to ten (AWS)

The app VMs depend on the database's hostname, so they wait for it. One `apply` from nothing is about ten minutes; `destroy` about six.

The two clouds, side by side: the network

Same shapes, different names. A configuration for one cloud is a translation exercise for the other, not a rewrite.

Piece	Azure	AWS
Network	`azurerm_virtual_network` + `azurerm_subnet`	`aws_vpc` + `aws_subnet`
Way out during the build	`azurerm_nat_gateway` + public IP + subnet association	`aws_nat_gateway` + `aws_eip` (elastic IP, AWS's public IP) + a route in the private route table
Firewall	`azurerm_network_security_group` + association	`aws_security_group`
Load balancer	`azurerm_lb` + backend pool + probe + rule	`aws_lb` + target group + listener + attachment

The two clouds, side by side: the machines

Piece	Azure	AWS
App VMs	`azurerm_linux_virtual_machine` + `azurerm_network_interface`	`aws_instance`
Database	`azurerm_postgresql_flexible_server`, delegated subnet, private DNS zone + link	`aws_db_instance` + `aws_db_subnet_group`
Bastion	a VM with `azurerm_public_ip`	an `aws_instance` in the public subnet

Whose rights does Terraform use?

`az login` makes Terraform *you*. On a student subscription you are **Owner**: every `apply` runs with the power to create, change or delete anything in the account, and so does every script that ever runs on the same laptop. That is fine for solo lab work, and it is exactly what does not scale: a deploy job in a pipeline, a teammate, a leaked key. Every one of those needs an identity of its own, with rights of its own. Both clouds have the machinery for that; they name it differently.

Access control: who may change what

Firewall rules control **packets**. Access control decides about **people and programs**: who may *create*, *change*, *delete* the infrastructure itself.

Three pieces, the same model in both clouds:

- **Identity**: a human user, or a machine identity: an account for a *program* that needs to do cloud things
- **Permission set**: a named bundle of allowed actions ("can read storage", "can restart VMs")
- **Scope**: the identity gets the permission set *on* something: one resource, a group of them, the whole account

Nothing in this model is cloud-specific. Each cloud built it under its own names.

The same model, two vocabularies

	Identity (machine)	Permission set	Scope
Azure	*service principal*, *managed identity*	**role** (Owner, Contributor, Reader, ...)	resource, resource group, subscription
AWS	*IAM user*, *IAM role*	**policy** (a JSON document of allowed actions)	the resources named in the policy, by ARN (Amazon Resource Name, a resource's id)

Azure calls the whole thing **RBAC** (role-based access control), under Microsoft Entra ID. AWS calls it **IAM** (Identity and Access Management). The vocabulary differs; the model is the same.

Terraform with an identity of its own

The same files, run by a program instead of a person:

```
# a service principal with Contributor on ONE resource group; credentials as environment variables
export ARM_CLIENT_ID=...  ARM_CLIENT_SECRET=...  ARM_TENANT_ID=...  ARM_SUBSCRIPTION_ID=...
terraform apply -auto-approve
```

- The azurerm provider reads those four variables instead of `az login`; the aws provider reads `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`, or an attached role
- Contributor on one resource group: the identity can build and destroy the setup, nothing outside it
- `-auto-approve` answers the *yes* a human would type; the plan was reviewed in an earlier step

Nothing in the `.tf` files changes between your laptop and the pipeline. That is why credentials stay out of them.

What Terraform doesn't solve

- **Drift**: `plan` detects it; the discipline of not clicking is yours. From now on the portal is for viewing, not changing
- **State on one laptop**: teammates cannot run apply, and a lost laptop is a lost state. Remote backends solve it, and this week's configuration does not use one
- **Copy and paste between projects**: the same network block in every configuration. Terraform's answer is modules; this week's configuration is one flat directory on purpose, so that every line is in view
- **Secrets in variables**: `db_password` travels in a `*.tfvars` file that must never be committed. That is a rule, not a mechanism

Today's scope: one configuration, understood line by line.

Assignment 3: what you build next

The A2 setup, built entirely by Terraform, and grown:

- `terraform apply` on an empty account builds it: network, two app VMs behind the load balancer, a managed database in its own private subnet, a NAT gateway, a bastion
- the validator goes green: A2's five checks, the beacons reaching the database, and, as the last step, your configuration uploaded to it. It must be Terraform, with no state file, no `*.tfvars`, no key in it
- `terraform apply -var locked=true` removes the NAT gateway; `terraform destroy` empties the account
- you start from the reference configuration: **filling in your values and understanding every block is the assignment**

Deadline: **Sunday of week 4, 23:59.** 2 points.

What to remember

- The portal is a client of an API; a `.tf` file is the same request, written down as **desired state**
- Terraform works with three things: your files, its **state**, and the cloud. Every run compares them
- Values flow between resources by **reference**, and references are the build order
- **State** is the link between a block and a cloud object: never edited by hand, never deleted, never committed
- `plan` before `apply`, always; `replaced` and `to destroy` are the words to slow down on
- A managed database is minutes to create and nothing to operate; apply it once
- One way in, the bastion; one way out, the NAT gateway, until `locked`



Questions