Gorgo Tasting OpenTofu and Terraform

September 08, 2026

Chapter 5

On the Internet

The need. When a system spans several vendors and several lifecycles, one configuration is no longer the correct unit. Infrastructure that changes rarely and an application that changes daily should be applied separately, with a way to pass values from one to the other. Values that should not be hard-coded, like the newest machine image, have to be looked up. Repeated resources need to be generated from a set rather than written out, and a configuration that reaches across a machine boundary has to be clear about where paths, commands, and secrets end up.

Why it matters. Separate root modules with remote state are how real projects limit the blast radius of a mistake and let people work in parallel. Data sources and `for_each` are what keep a configuration correct as images are retired and lists grow. Knowing where a provider acts, and what loses its sensitivity on the way, is the difference between a secret and a leak.

Why it is hard. A provider must be able to reach its target during plan, so anything the provider needs has to exist before the configuration that uses it, which forces the split. Cloud networks are many small resources that must agree, some of which already exist and have to be taken over rather than created. Credentials for three vendors have to stay out of the files, certificates expire, and destroying things in the wrong order strands a state.

The strategy. Two root modules: infra creates the machine, the DNS records, and the certificate; app reads infra's outputs through `terraform_remote_state` and deploys the Chapter 3 configuration as a module over Secure Shell (SSH). Data sources find the image, `for_each` and `dynamic` generate the records and the firewall rules, a heredoc holds the proxy configuration, and `sensitive()` protects what crosses between states. The example is the sayings server on an Oracle Cloud free tier machine, named through Cloudflare DNS and served over HTTPS with a Let's Encrypt certificate.

The providers

This chapter's two configurations live in `oracle/infra` and `oracle/app`, next to `modules`. Stage one, in `infra`, uses four providers, all new; stage two, in `app`, reuses Docker, random, and HTTP from Chapter 3. `oracle/infra/versions.tf`:

```
terraform {
  required_version = ">= 1.6"

  required_providers {
    oci = {
      source  = "oracle/oci"
      version = "~> 7.0"
    }
```

```
    cloudflare = {
      source  = "cloudflare/cloudflare"
      version = "~> 5.0"
    }
    acme = {
      source  = "vancluever/acme"
      version = "~> 2.0"
    }
    tls = {
      source  = "hashicorp/tls"
      version = "~> 4.0"
    }
  }
}

provider "oci" {
  # reads ~/.oci/config, written by `oci setup config`
}

provider "cloudflare" {
  # reads CLOUDFLARE_API_TOKEN
}

provider "acme" {
  server_url = var.acme_server
}
```

The **OCI** (Oracle Cloud Infrastructure) provider manages Oracle Cloud: networks, subnets, gateways, machines, and hundreds of other resource types, all prefixed `oci_`, plus data sources for looking things up, such as which machine images exist [1]. It authenticates with the same `~/.oci/config` file that Oracle's own command line tool writes and reads. The **Cloudflare** provider manages DNS zones and records, among much else, through Cloudflare's API and an API token [2]. The **ACME** (Automatic Certificate Management Environment) provider speaks the protocol Let's Encrypt uses to issue certificates: `acme_registration` creates an account and `acme_certificate` requests a certificate, proving control of the name through a DNS record that the provider creates and removes for you [3]. The **TLS** (Transport Layer Security) provider generates keys and certificates locally, the way `random` generates strings; this chapter uses `tls_private_key` for the ACME account key [4].

New syntax in this chapter

Five language features appear here for the first time or in a new role. Each is explained where it is used; this is the map.

- **Data sources**, the `data` blocks from Chapter 2's check, now do lookups: which availability domains the region has and which machine image is newest, values you should never hard-code.
- **`for_each`** on a resource creates one instance per element of a set or map, with `each.key` and `each.value` available inside and an address like `RESOURCE["key"]` for each instance; the DNS records use it.
- **`dynamic` over a map** is the Chapter 4 block generator fed a map instead of a list, so that each generated block sees `.key` and `.value`; the firewall rules use it.
- **Heredocs** (`<<-EOT ... EOT`) hold multi-line strings with interpolation, and the `-` strips the common indentation; the proxy configuration is one.
- **`terraform_remote_state`**, a built-in data source, reads another configuration's outputs, and **`sensitive()`** marks a value sensitive again when it lost that on the way; stage two uses both to consume stage one.

Accounts and credentials

Before any configuration, three things must be in place.

Oracle Cloud. Create a free tier account, install the `oci` command line tool, and run `oci setup config`. It writes `~/.oci/config` with your tenancy, user, region, and API key, and the provider reads that file. Find your tenancy OCID (Oracle Cloud Identifier) in the console under your profile; the tenancy is also the root compartment, and building in it is fine for a booklet.

Cloudflare. Your domain's DNS must be hosted by Cloudflare (the free plan is enough). Create an API token with the `Zone.DNS: Edit` permission for the zone, and copy the zone ID from the domain's overview page. Two things read the token, so export it under both names:

```
$ export CLOUDFLARE_API_TOKEN=...       # the cloudflare provider
$ export CLOUDFLARE_DNS_API_TOKEN=...   # the acme provider's dns challenge
```

Let's Encrypt. No account to create; the configuration registers one with an email address. Let's Encrypt has a staging environment for testing and strict rate limits on the real one, and the configuration starts on staging.

Trap: Everything in this chapter that is a secret comes from an environment variable or from `~/.oci/config`. The values that are not secret but are yours — the compartment, the domain, the zone ID — go in `terraform.tfvars`, which the examples' `.gitignore` excludes, with an `example.tfvars` committed to show the form. The one file you never want to see in `git status` is a `.tfvars` with a token in it.

Stage one: the infrastructure

The `infra` directory's providers are the four shown above. The variables are the things that are yours: `compartment_ocid`, `domain`, `cloudflare_zone_id`, `acme_email`, and with defaults `hostnames` (`["motd"]`), `ssh_public_key` (`~/.ssh/id_ed25519.pub`), `admin_cidr` (`0.0.0.0/0`, and you should narrow it), `ad_index` (`0`), and `acme_server` (the staging directory URL). See `variables.tf` in the examples for the descriptions and the validation. If you have no SSH key yet, `ssh-keygen -t ed25519` creates one in that place on macOS, Linux, and Windows alike.

The network

Cloud networking is a small building. The **VCN** (virtual cloud network) is the building. A **subnet** is a floor. The **internet gateway** is the front door, the **route table** is the signs pointing at it, and the **security list** is the guard's list of who may come in. `network.tf`:

```
resource "oci_core_vcn" "motd" {
  compartment_id = var.compartment_ocid
  display_name   = "motd"
  cidr_blocks    = ["10.0.0.0/16"]
  dns_label      = "motd"
}

resource "oci_core_internet_gateway" "motd" {
  compartment_id = var.compartment_ocid
  vcn_id         = oci_core_vcn.motd.id
  display_name   = "motd"
}

# every VCN comes with a default route table; take it over and add a route out
resource "oci_core_default_route_table" "motd" {
```

```
  manage_default_resource_id = oci_core_vcn.motd.default_route_table_id

  route_rules {
    destination       = "0.0.0.0/0"
    network_entity_id = oci_core_internet_gateway.motd.id
  }
}
```

Wut: Creating a VCN silently creates a default route table and a default security list with it. You cannot create them, because they already exist, and you cannot ignore them, because the subnet uses them. The `oci_core_default_*` resource types with `manage_default_resource_id` take over a default object and manage its contents. Several clouds have this pattern; the "default" resource types are the tell.

The security list is where `dynamic` from Chapter 4 pays off, because the rules follow the same pattern three times:

```
locals {
  open_ports = {
    ssh   = { port = 22, source = var.admin_cidr }
    http  = { port = 80, source = "0.0.0.0/0" }
    https = { port = 443, source = "0.0.0.0/0" }
  }
}

resource "oci_core_default_security_list" "motd" {
  manage_default_resource_id = oci_core_vcn.motd.default_security_list_id

  egress_security_rules {
    destination = "0.0.0.0/0"
    protocol    = "all"
  }

  # one ingress rule per entry in local.open_ports
  dynamic "ingress_security_rules" {
    for_each = local.open_ports
    content {
      protocol = "6" # tcp
      source   = ingress_security_rules.value.source
      tcp_options {
        min = ingress_security_rules.value.port
        max = ingress_security_rules.value.port
      }
    }
  }
}

resource "oci_core_subnet" "public" {
  compartment_id = var.compartment_ocid
  vcn_id         = oci_core_vcn.motd.id
  cidr_block     = "10.0.1.0/24"
  display_name   = "public"
  dns_label      = "public"
}
```

`open_ports` is a **map** of **objects**: keys to values, where each value has named fields. Iterating a map with `dynamic` gives `ingress_security_rules.value` (the object) and `ingress_security_rules.key` (the name), and adding a port is adding a line to the map.

The machine

`compute.tf` needs two lookups before it can create anything:

```
locals {
  shape = "VM.Standard.E2.1.Micro" # always free: 1 OCPU, 1 GB
  ads   = data.oci_identity_availability_domains.ads.availability_domains
}

data "oci_identity_availability_domains" "ads" {
  compartment_id = var.compartment_ocid
}

data "oci_core_images" "ubuntu" {
  compartment_id           = var.compartment_ocid
  operating_system         = "Canonical Ubuntu"
  operating_system_version = "24.04"
  shape                    = local.shape
  sort_by                  = "TIMECREATED"
  sort_order               = "DESC"
}
```

Data sources are the read-only lookups from Chapter 2, now doing real work: which availability domains the region has, and which image ids are current Ubuntu for this machine type. The image list is sorted newest first, so `images[0]` is the latest. Hard-coding an image id would break the month the image is retired; the lookup does not.

```
resource "oci_core_instance" "motd" {
  compartment_id      = var.compartment_ocid
  availability_domain = local.ads[var.ad_index].name
  display_name        = "motd"
  shape               = local.shape

  source_details {
    source_type = "image"
    source_id   = data.oci_core_images.ubuntu.images[0].id
  }

  create_vnic_details {
    subnet_id        = oci_core_subnet.public.id
    assign_public_ip = true
    hostname_label   = "motd"
  }

  metadata = {
    ssh_authorized_keys = file(pathexpand(var.ssh_public_key))
    user_data           = base64encode(file("${path.module}/cloud-init.yaml"))
  }
}
```

`metadata` is how the machine is told things at first boot. `ssh_authorized_keys` gets your public key: `pathexpand` turns `~/.ssh/...` into an absolute path and `file` reads it. `user_data` is a cloud-init document,

which Oracle wants base64 encoded, hence `base64encode(file(...))`.

`cloud-init.yaml` prepares the machine so that stage two only needs Docker:

```
#cloud-config
package_update: true
packages:
  - docker.io
runcmd:
  # the Oracle Ubuntu image ships a firewall that only lets SSH in
  - iptables -I INPUT 6 -m state --state NEW -p tcp --dport 80 -j ACCEPT
  - iptables -I INPUT 6 -m state --state NEW -p tcp --dport 443 -j ACCEPT
  - netfilter-persistent save
  # a little swap keeps a 1 GB instance alive while go builds
  - fallocate -l 2G /swapfile
  - chmod 600 /swapfile
  - mkswap /swapfile
  - swapon /swapfile
  - echo '/swapfile none swap sw 0 0' >> /etc/fstab
  # let the ubuntu user drive docker over ssh
  - usermod -aG docker ubuntu
  - systemctl enable --now docker
```

Trap: The security list opens ports 80 and 443 at the network, and then nothing answers. Oracle's Ubuntu images come with `iptables` rules that reject everything except SSH, inside the machine, and the console gives no hint. The two `iptables` lines in cloud-init are the fix, and `netfilter-persistent save` keeps them across reboots. This is the single most common "it does not work" in Oracle's free tier.

Trap: `Out of host capacity` means the availability domain has no free tier machines left, which is common. Try `-var ad_index=1`, or another region if the home region has one domain. The ARM machine type `VM.Standard.A1.Flex` is more generous (four cores and 24 GB free) but far harder to get; if you do get one, Docker builds the image for ARM on the machine itself, so nothing else changes.

The name

`dns.tf` points the names at the machine:

```
locals {
  fqdns = [for h in var.hostnames : "${h}.${var.domain}"]
}

resource "cloudflare_dns_record" "motd" {
  for_each = toset(local.fqdns)

  zone_id = var.cloudflare_zone_id
  name    = each.value
  type    = "A"
  content = oci_core_instance.motd.public_ip
  ttl     = 60
  proxied = false
}
```

`for_each` on a resource creates one instance per element of a set or map. Inside the block, `each.value` is the

element (and `each.key` its key, which for a set is the same thing). `toset` turns the list of names into a set, because `for_each` wants keys that identify each instance, not positions that shift when you insert one in the middle. The instances are addressed as `cloudflare_dns_record.motd["motd.example.com"]`.

`content` is the machine's public address, known only after the instance exists, which orders the record after the machine. `proxied = false` keeps Cloudflare out of the path so that the machine's own certificate is the one browsers see.

The certificate

`cert.tf`:

```
resource "tls_private_key" "account" {
  algorithm   = "ECDSA"
  ecdsa_curve = "P256"
}

resource "acme_registration" "motd" {
  account_key_pem = tls_private_key.account.private_key_pem
  email_address   = var.acme_email
}

resource "acme_certificate" "motd" {
  account_key_pem           = acme_registration.motd.account_key_pem
  common_name               = local.fqdns[0]
  subject_alternative_names = slice(local.fqdns, 1, length(local.fqdns))

  dns_challenge {
    provider = "cloudflare" # reads CLOUDFLARE_DNS_API_TOKEN
  }
}
```

ACME is the protocol Let's Encrypt speaks: prove you control the name, get a certificate for it. The **DNS challenge** proves control by publishing a token in a TXT record, which the provider does through Cloudflare and removes afterwards. It has two advantages over the HTTP challenge: nothing needs to be listening on port 80 yet, and it works for wildcard names.

The first name is the certificate's common name and the rest are alternative names; `slice(list, 1, length)` is "everything but the first".

Tip: Start on the staging server, which the `acme_server` default does. Staging certificates are not trusted by browsers, but they prove the whole chain works, and the production server limits you to five certificates per name per week. Once `https://motd.example.com/` answers (with a warning), apply once more with `-var acme_server=https://acme-v02.api.letsencrypt.org/directory` and the certificate is replaced with a real one.

Wut: A certificate expires after ninety days and the provider renews it by planning a replacement whenever thirty days or fewer remain (`min_days_remaining`). That means renewal is "run `tofu apply` at least monthly", by hand, by cron, or by a scheduled CI job. Nothing renews it while you are not looking.

Outputs

`outputs.tf` is what stage two needs to know:

```
output "public_ip" {
  value = oci_core_instance.motd.public_ip
}

output "fqdn" {
  value = local.fqdns[0]
}

output "certificate_pem" {
  value = acme_certificate.motd.certificate_pem
}

output "issuer_pem" {
  value = acme_certificate.motd.issuer_pem
}

output "private_key_pem" {
  value     = acme_certificate.motd.private_key_pem
  sensitive = true
}
```

Apply stage one with your values in `terraform.tfvars`:

```
$ cd oracle/infra
$ tofu init
$ tofu apply
```

Expect the machine to take a couple of minutes, cloud-init a couple more after that, and the DNS challenge a minute or two; unlike the timings in earlier chapters, these are estimates, since this stage was validated against the provider schemas but not applied while writing the booklet. When it finishes you can log in, and after cloud-init is done, `docker ps` works without `sudo`:

```
$ ssh ubuntu@$(tofu output -raw fqdn) docker ps
```

Stage two: the application

The `app` directory starts by reading stage one's outputs:

```
data "terraform_remote_state" "infra" {
  backend = "local"
  config = {
    path = "${path.module}/../infra/terraform.tfstate"
  }
}

locals {
  infra = data.terraform_remote_state.infra.outputs
}

provider "docker" {
  host     = coalesce(var.docker_host, "ssh://ubuntu@${local.infra.fqdn}")
  ssh_opts = ["-o", "StrictHostKeyChecking=accept-new"]
}
```

`terraform_remote_state` is a data source that reads another configuration's state file and exposes its outputs.

Only outputs: the other configuration decides what it publishes, and the rest of its state stays private. With a `local` backend the config is a path; with a shared backend it is the same settings the other configuration uses (Appendix A).

The Docker provider then points at the machine over SSH. The provider runs the same `ssh` you do (on Windows, the OpenSSH client that ships with the system), so your key and agent are used, and `accept-new` records the host key on first contact instead of stopping to ask, which would hang an apply. The `docker_host` variable defaults to `null` and exists so that the same configuration can be pointed at any Docker host, including your own for a dry run; `coalesce` returns its first argument that is not null.

Trap: The provider has to reach the machine during `plan`, not just `apply`. Run stage two before stage one is done and you get a connection error, not a plan. That is the reason for two stages: a provider's configuration has to be known, and reachable, before planning the things it manages.

Chapter 3 as a module

The Chapter 3 configuration moved into `modules/motd-docker` with four changes, and each teaches something about modules.

Values became variables. `app_dir`, `seed_sql`, `db_password`, `timezone`, and `port` are inputs now; the module builds the same network, volume, database, and web server from them.

The bind mount became an `upload`. Chapter 3 wrote the seed to a file and bind-mounted its directory into the database container. That file is on your laptop, and the Docker host is now a machine in Oracle's data center, where the path does not exist. The `upload` block copies content into the container at creation instead:

```
  upload {
    content = var.seed_sql
    file    = "/docker-entrypoint-initdb.d/001-sayings.sql"
  }
```

Wut: Paths in a `volumes` block are paths on the Docker host, not on the machine running `tofu`. That is invisible when they are the same machine and very visible when they are not. The same goes for `local-exec` provisioners, which run where `tofu` runs, and `remote-exec` provisioners, which run on the resource. Always ask "where does this path live" when a configuration crosses a machine boundary.

A resource stands in for the seed. `replace_triggered_by` only accepts resource references, and there is no `local_file` any more, so the module wraps the SQL in a `terraform_data` and the volume watches that:

```
resource "terraform_data" "seed" {
  input = var.seed_sql
}

resource "docker_volume" "db_data" {
  name = "motd-db-data"

  lifecycle {
    replace_triggered_by = [terraform_data.seed]
  }
}
```

The port became optional. A `dynamic "ports"` block over `var.port == null ? [] : [var.port]` publishes the web server's port only when asked, the Chapter 4 idiom. Stage two does not ask, because only the proxy should talk to the web server.

The module's outputs are the network name and the web container's name, which are the two things the proxy needs.

The proxy

`main.tf` in `app`:

```
module "sayings" {
  source = "github.com/BooksByGorgo/opentofu//tasting/examples/motd?ref=main"
}

resource "random_password" "db" {
  length  = 24
  special = false
}

module "motd" {
  source = "../../modules/motd-docker"

  app_dir     = "${path.module}/app"
  seed_sql    = module.sayings.sql
  db_password = random_password.db.result
  timezone    = var.timezone
  # no host port: only the proxy talks to the web server
}

locals {
  caddyfile = <<-EOT
    ${local.infra.fqdn} {
        tls /certs/tls.crt /certs/tls.key
        reverse_proxy ${module.motd.web_container}:8080
    }
  EOT
}
```

The Caddyfile is a **heredoc**: `<<-EOT` starts a multi-line string that ends at a line containing only `EOT`, and the `-` strips the common leading indentation so that the configuration can stay indented. Interpolation works inside it. Caddy is told the name it serves, where its certificate is, and where to send requests; given a name and a certificate it serves HTTPS on 443 and redirects port 80 by itself.

```
resource "docker_image" "caddy" {
  name = "caddy:2"
}

resource "docker_container" "proxy" {
  name  = "motd-proxy"
  image = docker_image.caddy.image_id

  networks_advanced {
    name = module.motd.network
  }

  ports {
    internal = 80
    external = var.http_port
```

```
  }

  ports {
    internal = 443
    external = var.https_port
  }

  upload {
    content = local.caddyfile
    file    = "/etc/caddy/Caddyfile"
  }

  upload {
    content = "${local.infra.certificate_pem}${local.infra.issuer_pem}"
    file    = "/certs/tls.crt"
  }

  upload {
    content = sensitive(local.infra.private_key_pem)
    file    = "/certs/tls.key"
  }
}
```

Three uploads: the Caddyfile, the certificate with its issuer chain appended, and the private key. A changed upload replaces the container, so a renewed certificate in stage one becomes a new proxy in stage two.

Trap: Outputs lose their sensitivity when read through `terraform_remote_state`. Without the `sensitive()` wrapper, the private key is printed in full in every plan that creates the proxy, and in the logs of whatever CI ran it. Wrap any secret that comes from another state, and check the plan for anything that should say `(sensitive value)` and does not.

The check is the one from Chapter 3 with the real URL and `insecure = true`, because staging certificates are not trusted by the HTTP provider either.

Running it

```
$ cd ../app
$ tofu init
$ tofu apply
```

The first apply builds the image on the machine, which should take a few minutes on one core with a gigabyte of memory (the swap in cloud-init is what makes it possible), pulls MySQL and Caddy, loads the sayings, and starts the proxy. That estimate is untested too: the application stage was applied against a local Docker daemon with a self-signed certificate standing in for stage one, not against the Oracle machine. Then:

```
$ curl -k $(tofu output -raw url)
```

Or open the URL in a browser, accept the staging warning, and read your saying. Switch stage one to the production ACME server, apply it, apply stage two, and the warning is gone.

Trap: Destroy in the reverse order of creation: `app` first, then `infra`. Destroying `infra` first deletes the machine that `app`'s Docker provider needs to reach, and `tofu destroy` in `app` then fails trying to refresh containers that no longer exist. If it happens, `tofu destroy -refresh=false` in `app` or deleting its state gets you unstuck; the containers died with the machine.

Key Points

- Split configurations by lifecycle: infrastructure that changes rarely, application that changes often. `terraform_remote_state` passes outputs from one to the other.
- Cloud networks are buildings: VCN, subnet, gateway, route table, security list. Default objects are managed with the `default` resource types.
- Data sources look up things you should not hard-code, like the newest image.
- cloud-init prepares a machine at first boot, and on Oracle it must also open the machine's own firewall.
- `for_each` makes one resource instance per element of a set or map, addressed by key.
- The ACME DNS challenge issues certificates without needing anything listening, and renewal means applying again once thirty days or fewer remain.
- Paths and commands run where the provider's target is, not where you are; `upload` moves content across that boundary.
- Secrets read from another state are not sensitive until you say so.

New Syntax

Syntax	What it is
`data "TYPE" "NAME" { filters }`	A lookup, with results in `data.TYPE.NAME.ATTR`
`for_each = toset(list)` with `each.value`	One resource instance per element
`RESOURCE["key"]`	Address of one instance of a `for_each` resource
`{ a = { port = 22 } }`	A map of objects
`dynamic` over a map: `.key`, `.value`	Iterating a map into nested blocks
`<<-EOT ... EOT`	Indented heredoc string
`file`, `pathexpand`, `base64encode`, `toset`, `slice`	More built-in functions
`data "terraform_remote_state"`	Read another configuration's outputs
`sensitive(x)`	Mark a value sensitive
`manage_default_resource_id`	Take over a cloud's default object
`upload { content, file }`	Copy content into a container at creation

Try It

- Add `"www"` to `hostnames` and apply both stages. Which resources change in each?
- Remove the certificate plumbing and let Caddy get its own certificate: a Caddyfile with just the name and `reverse_proxy` does it, using the HTTP challenge on port 80. Compare the two approaches: what does each need open, and who renews?
- Move stage one's state to Oracle Object Storage using the `s3` backend with the compatibility endpoint, and change stage two's remote state to match.
- Write a GitHub Actions workflow that runs `tofu apply` on stage one weekly to renew the certificate, with the tokens in repository secrets.
- Get an `A1.Flex` machine, install k3s from cloud-init, fetch its kubeconfig, and deploy the Chapter 4 module to it with a `kubernetes_ingress_v1` that uses the certificate as a `kubernetes.io/tls` secret. You will need to push the image to a registry; `ghcr.io` with a GitHub token that has the `write:packages` scope works.
- Narrow `admin_cidr` to your own address and confirm SSH from elsewhere fails and HTTPS still works.

Exercises

1. **Think about it:** Stage two reads stage one's outputs through remote state. Why not put everything in one configuration, with the Docker provider's `host` built from `oci_core_instance.motd.public_ip`

by interpolation?

2. **What does this do?** With `hostnames = ["motd", "www"]` and `domain = "example.com"`, what are the addresses of the DNS record instances, and what are `common_name` and `subject_alternative_names` on the certificate?
3. **Calculation:** A certificate was issued on March 1st with `min_days_remaining = 30`. On which day does `tofu plan` first show it being replaced, and if you apply only on the first of each month, how many days of validity does the old certificate have left on the day it is replaced?
4. **Where is the bug?**

```
resource "cloudflare_dns_record" "motd" {
  for_each = local.fqdns

  zone_id = var.cloudflare_zone_id
  name    = each.value
  type    = "A"
  content = oci_core_instance.motd.public_ip
  ttl     = 60
}
```

5. **Where is the bug?** The machine is up, the security list opens 443, DNS resolves, Caddy is running, and `curl https://motd.example.com/` times out.
6. **Write a configuration** for a second, staging machine in the same VCN with its own name (`motd-staging.example.com`), sharing the subnet and security list, by turning the instance, the record, and the certificate into a module called twice. Decide what the module's variables should be before you write it.

[1] Oracle, "Terraform provider for oracle cloud infrastructure." 2026. Available: https://docs.oracle.com/en-us/iaas/Content/dev/terraform/home.htm

[2] Cloudflare, "Terraform provider for cloudflare." 2026. Available: https://github.com/cloudflare/terraform-provider-cloudflare/blob/master/docs/index.md

[3] C. Marchesi, "Terraform provider for ACME: documentation." 2026. Available: https://github.com/vancluever/terraform-provider-acme/blob/main/docs/index.md

[4] HashiCorp, "Terraform provider for TLS: documentation." 2026. Available: https://github.com/hashicorp/terraform-provider-tls/blob/main/docs/index.md