Gorgo Tasting OpenTofu and Terraform — Exercise Answers

September 08, 2026

1. Hello, World

1. Think about it: `tofu apply` prints hello world the first time and nothing the second time. Explain what OpenTofu compared to decide there was nothing to do, and where each side of the comparison came from.

Answer: It compared the desired state with the recorded state. The desired state comes from the configuration files: one `terraform_data.hello` should exist. The recorded state comes from `terraform.tfstate`, written by the first apply: a `terraform_data.hello` exists with a given id. The two agree, so the plan is empty and no provisioner runs. For resources backed by a provider, OpenTofu also refreshes the recorded state against reality before comparing, which is how it notices things deleted behind its back.

---

2. What does this do? Given this configuration and a fresh directory, what does `tofu apply -auto-approve` print, in order?

```
resource "terraform_data" "first" {
  provisioner "local-exec" {
    command = "echo one"
  }
}

resource "terraform_data" "second" {
  input = terraform_data.first.id

  provisioner "local-exec" {
    command = "echo two"
  }
}
```

Answer: `one`, then `two`. `second` references `terraform_data.first.id`, so `first` must be created before `second`, and each provisioner runs when its resource is created. Without the reference, the two could be created in either order or at the same time.

---

3. Calculation: A configuration has three `terraform_data` resources, each with `triggers_replace = [var.tag]`, and the state was applied with `tag = "a"`. What is the plan summary line for `tofu plan -var tag=b`?

Answer: `Plan: 3 to add, 0 to change, 3 to destroy.` Every resource's trigger changed, so each one is destroyed and recreated, which counts as one destroy and one add.

---

4. Where is the bug?

```
variable "name" {
  type    = string
  default = "gorgo"
}

resource "terraform_data" "greet" {
  provisioner "local-exec" {
    command = "echo hello var.name"
  }
}
```

Answer: The command prints `hello var.name`. Inside a string, a reference has to be interpolated: `"echo hello ${var.name}"`. Without the `${...}`, `var.name` is just text.

---

5. Write a configuration with a variable `path` that must end in `.txt` (use `endswith`), and a resource that writes the current date into that file with `date > path`. Make sure a second apply does not rewrite the file, and that changing `path` does.

Answer:

```
variable "path" {
  type = string

  validation {
    condition     = endswith(var.path, ".txt")
    error_message = "path must end in .txt."
  }
}

resource "terraform_data" "stamp" {
  triggers_replace = [var.path]

  provisioner "local-exec" {
    command = "date > '${var.path}'"
  }
}
```

The second apply finds the resource already in state and does nothing. Changing `path` changes `triggers_replace`, so the resource is replaced and the provisioner writes the new file. The old file is not removed; a `provisioner "local-exec" { when = destroy ... }` could do that.

2. A Web Server on Your Machine

1. Think about it: The image resource has a `triggers` map holding a hash of the source directory. What would happen on the second `apply` if the hash were replaced with `timestamp()`? What would happen if `triggers` were removed entirely and you edited `main.go`?

Answer: With `timestamp()` the trigger value changes on every plan, so every apply rebuilds the image and replaces the container, even when nothing changed. That is the "provisioner on every run" anti-pattern in a different coat. Without `triggers`, editing `main.go` changes nothing OpenTofu can see: the image name is still `motd:ch2`, so the plan is empty and the old server keeps running.

---

2. What does this do? For the expression below, what is the type of the result and how many elements does it have when `app/` contains `main.go`, `go.mod`, and `Dockerfile`?

```
[for f in fileset("app", "*.go") : upper(f)]
```

Answer: A list of strings with one element, `["MAIN.GO"]`. `fileset` matches only `main.go` against `*.go`, and the for expression applies `upper` to each match. Since the input is a set, the result is a list built in lexical order of the set's elements.

---

3. Calculation: A configuration pins `version = "~> 3.4"` for a provider whose available releases are 3.3.0, 3.4.2, 3.9.0, and 4.0.1. Which release does `tofu init` install, and which is the newest it could ever install without editing the constraint?

Answer: `~> 3.4` means `>= 3.4, < 4.0`. `init` installs the newest that fits, 3.9.0. The newest it could ever install is the last 3.x release; 4.0.1 is excluded. Once the lock file exists, `init` keeps 3.9.0 until you run `init -upgrade`.

---

4. Where is the bug?

```
resource "docker_container" "web" {
  name  = "motd"
  image = docker_image.web.name

  ports {
    internal = 8080
    external = var.port
  }
}
```

Answer: `image` refers to the image's `name`, which is the fixed string `motd:ch2`. The container works the first time, but when the source changes and the image is rebuilt, the name is the same, so the container is not replaced and keeps running the old binary. Use `docker_image.web.image_id`, which changes with every build.

---

5. Where is the bug?

```
locals {
  app_dir = "${path.module}/app"
}

resource "docker_image" "web" {
  name = "motd:ch2"
  build {
    context = locals.app_dir
  }
}
```

Answer: The block is `locals` (plural) but the reference is `local.app_dir` (singular). `locals.app_dir` is a reference to an undeclared resource named `locals`, and `tofu validate` says so.

---

6. Write a configuration that runs two copies of the web server on ports 8080 and 8081 from the same image, with a check for each. Then reduce the duplication with a variable of type `list(number)` and `count` (look it up), and compare the plans.

Answer: The duplicated version has two `docker_container` blocks, `web_a` and `web_b`, differing only in `name` and `external`, and two `check` blocks. The `count` version:

```
variable "ports" {
  type    = list(number)
  default = [8080, 8081]
}

resource "docker_container" "web" {
  count = length(var.ports)
  name  = "motd-${count.index}"
  image = docker_image.web.image_id

  ports {
    internal = 8080
    external = var.ports[count.index]
  }
}
```

The instances are addressed `docker_container.web[0]` and `docker_container.web[1]`. The plans create the same containers; the difference shows up when you remove the first port: with `count`, the container at index 0 is replaced to take the other port and the one at index 1 is destroyed, instead of one container simply going away, which is the reason `for_each` (Chapter 5) is usually preferred. A `check` block cannot use `count`, so the check either loops in one assertion with `alltrue([for c in docker_container.web : ...])` or a data source with `for_each`.

3. A Database of Sayings

1. Think about it: The database container has `wait = true` and the web container has `depends_on = [docker_container.db]`. What would go wrong if you removed only `wait`? What if you removed only `depends_on`?

Answer: Without `wait`, the database counts as created the moment the container starts, so the web server starts while MySQL is still loading the seed. The server itself copes (it answers 503 until the database answers), but the `check` may run before the seed is loaded and warn. Without `depends_on`, nothing orders the web container after the database at all; they start together, with the same result, and on a destroy the database may be removed before the web server, which does not matter here but would for anything that needed to flush.

---

2. What does this do? What does the template below produce for `names = ["a", "b", "c"]`?

```
%{ for i, n in names ~}
${i}:${n}${i < length(names) - 1 ? "," : ""}
%{ endfor ~}
```

Answer:

```
0:a,
1:b,
2:c
```

Each iteration emits the index, a colon, the name, and a comma except on the last; the `~` on the directives eats the newlines after them, so the output has exactly one line per name.

---

3. Calculation: At 14:05:09 local time, which saying does the server return, as a prefix index and a suffix index? Which second of which minute returns `module.sayings.sayings[3599]`?

Answer: The key is `5 * 60 + 9 = 309`, which is prefix `309 / 60 = 5` and suffix `309 % 60 = 9`: "A careful reviewer asks why before asking how." Key 3599 is minute 59, second 59: `59 * 60 + 59`.

---

4. Where is the bug?

```
resource "docker_container" "db" {
  name  = "motd-db"
  image = docker_image.mysql.image_id

  volumes {
    host_path      = "./seed"
    container_path = "/docker-entrypoint-initdb.d"
  }
}
```

Answer: Docker requires an absolute host path for a bind mount, and `./seed` is relative. Use `abspath("${path.module}/seed")`. There is also no dependency on whatever writes the seed, so the container could start before the file exists; referencing `local_file.seed.filename` fixes both.

---

5. Where is the bug?

```
output "dsn" {
  value = "motd:${random_password.db.result}@tcp(db:3306)/motd"
}
```

Answer: The value contains a sensitive value, and an output that does is required to be marked `sensitive = true`; the plan fails with `Output refers to sensitive values`. Add the marking, and think about whether an output that contains a password should exist at all.

---

6. Write a configuration that adds a second table, `visits`, seeded from a template with one row per weekday, and a second `check` that queries the database through `docker exec` in a `terraform_data` provisioner. Decide whether the provisioner or an `http` data source is the better tool, and say why.

Answer: The seed is a second `local_file`, `seed/002-visits.sql`, rendered from a template over `["Monday", ..., "Friday"]`, in the same directory so MySQL runs it after `001-sayings.sql`. The query can be a `terraform_data` with `triggers_replace = [docker_container.db.id]` and a `local-exec` running `docker exec motd-db mysql -umotd -p... motd -e 'SELECT COUNT(*) FROM visits'`. That works but is the wrong tool: a provisioner runs only at creation, its result is not in the state, and a failure taints the resource. A `check` block can only hold data sources, and there is no data source that runs SQL through docker; the better design is to expose `/healthz` on the web server that runs the query, and check that with `data "http"`. Checks should test through the application's own interface anyway.

4. Kubernetes

1. Think about it: The image tag is derived from the source hash instead of using `motd:latest` with `image_pull_policy = "Always"`. Give two things that would go wrong with the `latest` approach in this configuration.

Answer: First, the deployment's `image` argument never changes, so the plan is empty after a rebuild and Kubernetes has no reason to roll out anything; you would have to delete pods by hand. Second, `Always` makes the cluster pull on every pod start, and kind has no registry to pull from, so pods fail with `ImagePullBackOff` even though the image was loaded. A third: with `latest` there is no way to tell from `kubectl get pods` which source is running.

---

2. What does this do? How many `ports` blocks does this produce when `var.ports = [80, 443]`, and what does `ports.value` refer to in each?

```
dynamic "ports" {
  for_each = var.ports
  content {
    internal = ports.value
    external = ports.value
  }
}
```

Answer: Two `ports` blocks. In the first, `ports.value` is `80`; in the second, `443`. The iterator is named after the block (`ports`) unless an `iterator` argument renames it.

---

3. Calculation: The web deployment has `replicas = 2`, and each pod's readiness probe runs every 5 seconds. Roughly how long after the database becomes ready can the deployment be reported created, at the earliest? What in the configuration bounds the worst case?

Answer: At the earliest, one probe period after the database answers, about five seconds, since both pods probe independently and can pass on the same cycle. In practice add the initial delay and a probe or two, so ten to fifteen seconds. The worst case is bounded by `timeouts { create = "5m" }` on the deployment, after which apply fails.

---

4. Where is the bug?

```
module "motd" {
  source      = "../modules/motd-k8s"
  image       = docker_image.web.name
  db_password = random_password.db.result
  node_port   = 30080
}
```

Answer: `seed_sql` has no default in the module and is not given, so `validate` fails with `Missing required argument`. The missing `depends_on = [terraform_data.kind_load]` is a second, quieter bug: the pods may start before the image is loaded into the cluster, and the rollout waits on `ImagePullBackOff` until it times out.

---

5. Where is the bug?

```
env {
  name  = "DB_DSN"
  value = "motd:${DB_PASSWORD}@tcp(db:3306)/motd"
}
```

Answer: `${DB_PASSWORD}` is an OpenTofu interpolation, and there is no `DB_PASSWORD` in the configuration, so validation fails with a reference to an undeclared resource. The Kubernetes expansion syntax uses parentheses: `$(DB_PASSWORD)`, which the configuration language passes through untouched.

---

6. Write a module around `kubernetes_resource_quota_v1`: name it `namespace-quota`, give it a namespace name and a maximum number of pods as variables, and have it create the namespace with that quota. Call it from the root for two namespaces and confirm `kubectl describe quota` in each.

Answer:

```
variable "name" { type = string }
variable "max_pods" { type = number }

resource "kubernetes_namespace_v1" "this" {
  metadata {
    name = var.name
  }
}

resource "kubernetes_resource_quota_v1" "pods" {
  metadata {
    name      = "pods"
    namespace = kubernetes_namespace_v1.this.metadata[0].name
  }
  spec {
    hard = {
      pods = var.max_pods
    }
  }
}
```

The root calls it twice:

```
module "team_a" {
  source   = "./namespace-quota"
  name     = "team-a"
  max_pods = 5
}

module "team_b" {
  source   = "./namespace-quota"
  name     = "team-b"
  max_pods = 10
}
```

`kubectl describe quota -n team-a` shows `pods 0 5`. With `for_each = { team-a = 5, team-b = 10 }` on a single `module` block, the two calls collapse into one with `each.key` and `each.value`.

5. On the Internet

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?

Answer: The provider's configuration must be known and usable at plan time, and on the first run the instance does not exist, so the host is `(known after apply)` and the provider cannot connect to plan the containers. Even after the first apply, replacing the instance would put the provider in the same position. Separate configurations also separate the lifecycles: the application is applied many times a day, the machine and the certificate rarely, and a mistake in one cannot destroy the other.

---

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?

Answer: `local.fqdns` is `["motd.example.com", "www.example.com"]`. The records are `cloudflare_dns_record.motd["m` and `cloudflare_dns_record.motd["www.example.com"]`. The certificate has `common_name = "motd.example.com"` and `subject_alternative_names = ["www.example.com"]`, from `slice(local.fqdns, 1, 2)`.

---

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?

Answer: Let's Encrypt certificates last 90 days, so it expires on May 30th. Thirty days or fewer remain from April 30th, and the provider renews at thirty, so a plan on April 30th first shows the replacement (a plan on April 29th sees 31 days). Applying on the first of each month, April 1st is too early (59 days left) and May 1st does it, with 29 days left on the old certificate.

---

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
}
```

Answer: `local.fqdns` is a list, and `for_each` accepts only a set of strings or a map; the error is `The given "for_each" argument value is unsuitable`. Wrap it: `for_each = toset(local.fqdns)`. A list would also be a poor choice even if it were allowed, because the instances would be keyed by position.

---

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.

Answer: A timeout, rather than a refusal or a TLS error, means packets are being dropped, and the usual culprit is the firewall inside the machine. Oracle's Ubuntu image rejects everything but port 22 with `iptables`, so cloud-init must insert the accept rules for 80 and 443 and save them. Check with `sudo iptables -L INPUT -n --line-numbers` on the machine; if the rules are missing, cloud-init did not run the `runcmd` section, and `/var/log/cloud-init-output.log` says why.

---

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.

Answer: The module needs what differs between the two machines and what they share but cannot look up: `name` (used for the display name and the host label), `hostnames`, `domain`, `zone_id`, `subnet_id`, `availability_domain`, `image_id`, `compartment_ocid`, `ssh_public_key`, `acme_email`, and the `account_key_pem` of one shared ACME registration. The module holds the `oci_core_instance`, the `cloudflare_dns_record` with `for_each`, and the `acme_certificate`, and outputs the public IP, the fully qualified name, and the certificate parts. The root keeps the VCN, subnet, gateway, route table, security list, the data sources, and one `acme_registration`, and calls the module as `module "prod"` and `module "staging"` with different `name` and `hostnames`. Stage two then reads `module.prod` or `module.staging` outputs from the root's outputs, which is a hint that stage two should take a variable naming the environment.