Terraform – Design Pattern

Service-Based Structure:

- Organizes resources by services or components (e.g., microservices).
- Each service has its own directory with infrastructure definitions.
- Encourages isolation but may lead to circular dependencies.

Module-Based Structure:

- Uses Terraform modules to abstract and reuse infrastructure code.
- Enforces consistency and scalability across environments.
- Separates infrastructure code from service code.

Terraform – Design Pattern – Single file

```
0.example.com, web-1.example.com, ...

variable "disk_image" {}
variable "public_key" {}

resource "google_compute_instance" "web" {
  count        = "10"
  name         = "web-${count.index}"
  zone         = "europe-west1-b"
  tags         = ["docker", "no-ip"]
  machine_type = "n1-standard-1"

  disk {
    image = "${var.disk_image}"
  }
  metadata {
    sshKeys = "${var.public_key}"
  }
  network_interface {
    network = "default"
  }
```

Terraform – Design Pattern – Multiple File

As your infrastructure needs grow bigger, so does your Terraform file. To alleviate this maintenance burden we can group common resources together into separate files, because Terraform reads all the `.tf` files in the directory. We could move to a structure like this for example:

```
vm.tf
firewall.tf
routes.tf
dns.tf
```

Splitting on resource type can be helpful, but it does make it harder to look at the logical pieces in your infrastructure. For instance, if we want to find all the resources relating to our database servers then we'd have to dig through all the files and look for relevant definitions. We could split the resources out logically instead:

```
database.tf
web.tf
site.tf # common resources like DNS zones
```

...but then we end up with the opposite problem and have resource types scattered around multiple files.

Terraform – Design Pattern - Modules

```
# Shared resources go here. Networks, dns zones, ...
modules/site/network.tf
modules/site/dns.tf
modules/site/variables.tf

# Each logical component becomes a module
modules/database/vm.tf
modules/database/dns.tf
modules/database/network.tf
modules/database/variables.tf

modules/web/vm.tf
modules/web/dns.tf
modules/web/network.tf
modules/web/variables.tf

# Leaving the root relatively clean
site.tf
variables.tf
terraform.tfvars
```

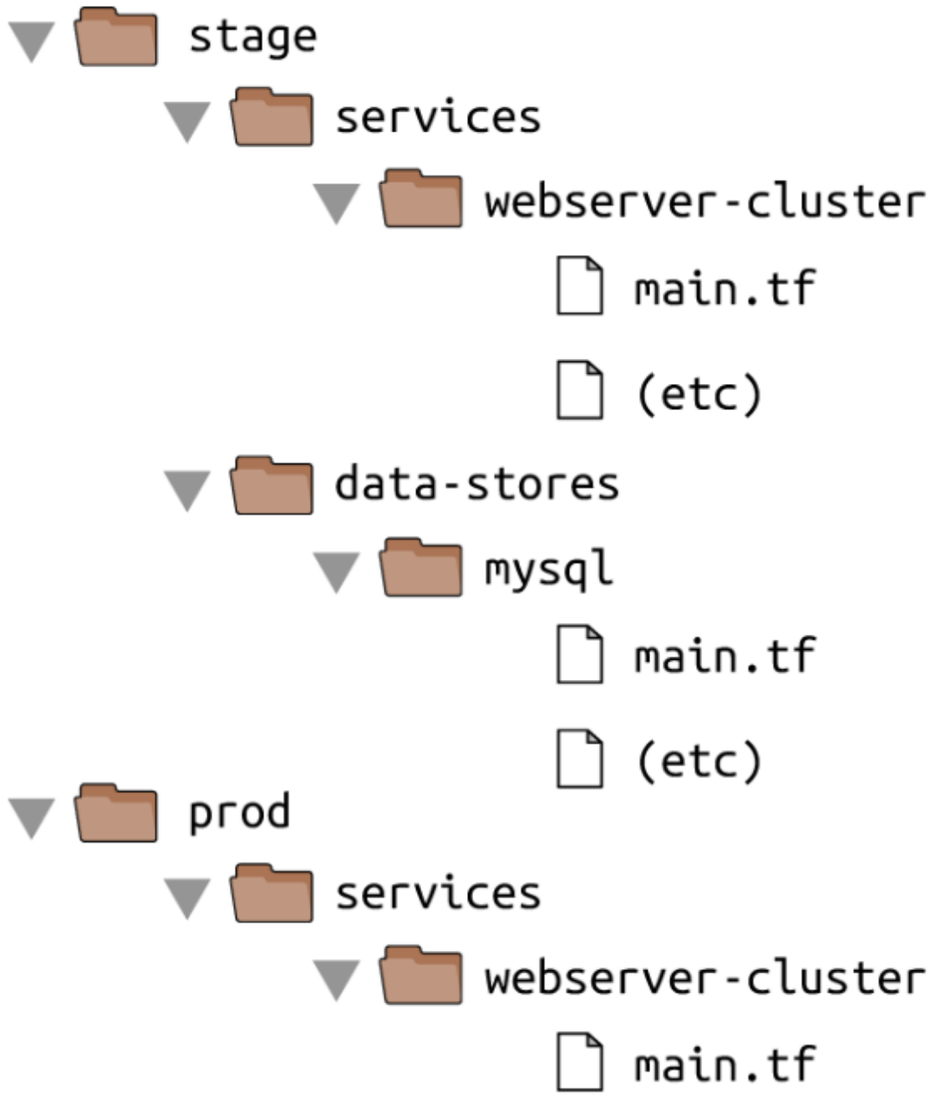
Terraform – Design Pattern – Pattern Modules

```
modules/single_instance/vm.tf
modules/single_instance/dns.tf
modules/single_instance/variables.tf

modules/load_balanced_pool/vm.tf
modules/load_balanced_pool/dns.tf
modules/load_balanced_pool/variables.tf
```

But the resources defined inside these files are fairly abstract and depend almost entirely on configuration. Below is an example of what the `single_instance` module could look like:

```
variable "name"          {}
variable "zone"          {}
variable "tags"          {}
variable "machine_type"  {}
variable "disk_image"    {}
variable "public_key"    {}
variable "network"       {}
variable "dns_zone_name" {}
variable "dns_domain"    {}

resource "google_compute_instance" "single_instance" {
  name         = "${var.name}"
  zone         = "${var.zone}"
  tags         = "${split(";", var.tags)}"
  machine_type = "${var.machine_type}"
```

Terraform – Design Pattern - Composing Pattern Modules



Terraform – Best Practices – Use Remote State

It's ok to use the local state when experimenting, but use a remote shared state location for anything above that point. Having a single remote backend for your state is considered one of the first best practices you should adopt when working in a team. Pick one that supports **state locking** to avoid multiple people changing the state simultaneously. Treat your state as immutable and avoid manual state changes at all costs. Make sure you have backups of your state that you can use in case of a disaster. For some backends, like AWS S3, you can enable *versioning* to allow for quick and easy state recovery.

Terraform – Best Practices –
Use existing shared and community modules

Instead of writing your own modules for everything and reinventing the wheel, check if there is already a module for your use case. This way, you can save time and harness the power of the Terraform community. If you feel like it, you can also help the community by improving them or reporting issues. You can check the Terraform Registry for available modules.

Terraform – Best Practices – Import existing infrastructure

If you inherited a project that is a couple of years old, chances are that some parts of its infrastructure were created manually. Fear not, you can import existing infrastructure into Terraform and avoid managing infrastructure from multiple endpoints.

Terraform – Best Practices – Avoid variables hard-coding

It might be tempting to hardcode some values here and there but try to avoid this as much as possible. Take a moment to think if the value you are assigning directly would make more sense to be defined as a variable to facilitate changes in the future. Even more, check if you can get the value of an attribute via a *data source* instead of setting it explicitly. For example, instead of finding our AWS account id from the console and setting it in *terraform.tfvars* as

```
aws_account_id="99999999999"
```

Terraform – Best Practices –
Always format and validate

In IaC, consistency is essential long-term, and Terraform provides us with some tools to help us in this quest. Remember to run **terraform fmt** and **terraform validate** to properly format your code and catch any issues that you missed. Ideally, this should be done auto-magically via a CI/CD pipeline or pre-commit hooks.

Terraform – Best Practices – Use a consistent naming convention

You can find online many suggestions for naming conventions for your Terraform code. The most important thing isn't the rules themselves but **finding a convention that your team is comfortable with** and trying collectively to be consistent with it. If you need some guidance, here's a list of rules that are easy to follow:

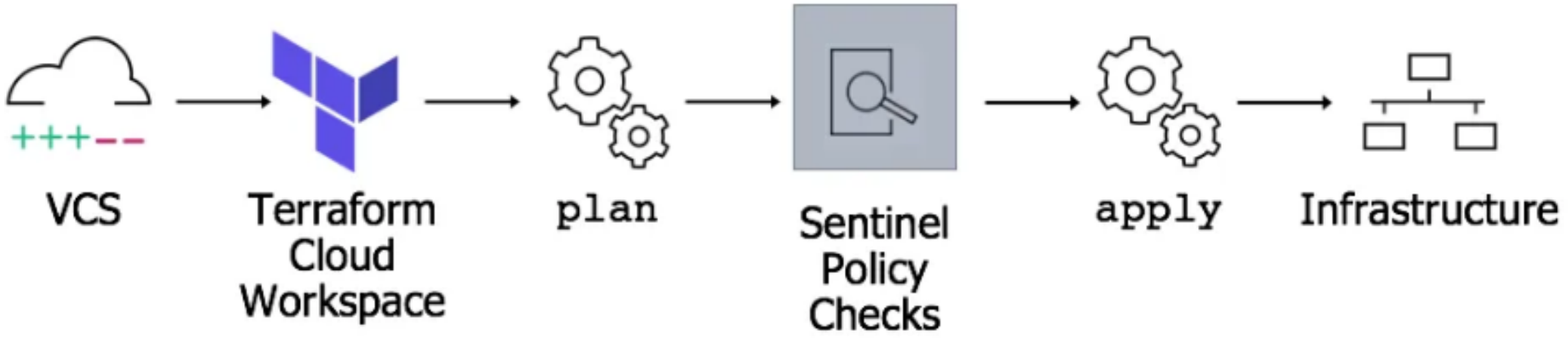
- Use *underscores*(_) as a separator and *lowercase letters* in names.
- Try not to repeat the resource type in the resource name.
- For single-value variables and attributes, use *singular nouns.* For lists or maps, use *plural nouns* to show that it represents multiple values.
- Always use descriptive names for variables and outputs, and remember to include a *description*.

Terraform – Best Practices – Tag your Resources

A robust and consistent tagging strategy will help you tremendously when issues arise or trying to figure out which part of your infrastructure exploded your cloud vendor's bill. You can also craft some nifty access control policies based on tags when needed. Like when defining naming conventions, try to be consistent and always tag your resources accordingly.

The Terraform argument **tags** should be declared as the last argument(only *depends_on* or *lifecycle*

Terraform – Best Practices – Introduce Policy as Code



How Sentinel fits into Terraform Cloud runs

Terraform – Best Practices – Implement a Secrets Management Strategy

Usually, users won't admit that they have secrets in their Terraform code, but we have all been there. When you are starting with Terraform, it's normal that secret management isn't your top priority, but eventually, you will have to define a strategy for handling secrets.

As you probably heard already, **never store secrets in plaintext and commit them in your version control system**. One technique that you can use is to pass secrets by setting *environment variables* with **TF_VAR** and marking your sensitive variables with **sensitive = true**.

Terraform – Best Practices – Test your Terraform code

As with all other code, IaC code should be tested properly. There are different approaches here, and again, you should find one that makes sense for you. Running *terraform plan* is the easiest way to verify if your changes will work as expected quickly. Next, you can perform some static analysis for your Terraform code without the need to apply it. Unit testing is also an option to verify the normal operation of distinct parts of your system.

Another step would be to integrate a Terraform linter to your CI/CD pipelines and try to catch any possible errors related to Cloud Providers, deprecated syntax, enforce best practices, etc. One step ahead, you can set up some integration tests by spinning up a replica sandbox environment, applying your plan there, verifying that everything works as expected, collecting results, destroying the sandbox, and moving forward by applying it to production.

Terraform – Best Practices
Enable debug/troubleshooting

When issues arise, we have to be quick and effective in gathering all the necessary information to solve them. You might find it helpful to set the Terraform log level to *debug* in these cases.

```
TF_LOG=DEBUG <terraform command>
```

Another thing that you might find helpful is to persist logs in a file by setting the *TF_LOG_PATH*

Terraform – Best Practices
Build modules wherever possible

If there is no community module available for your use case, it is encouraged to build your own module. Even though sometimes, you will start by building something that seems trivial, as your Infrastructure matures, you will need to come back to your simple module and add more features to it.

Of course, this will also help when you have to replicate your code in another environment,, as all you will need to do is create an object from that module and populate it with the correct parameters for the new environment.

Terraform – Best Practices

Use loops and conditionals

Your code should be able to create multiple instances of a resource whenever possible, so a recommendation would be to use count or for_each on the ones that may change from one environment to the other. This, combined with conditionals, will give you the flexibility to accommodate many different use cases with the same code and offer genericity to your parameters.

Terraform – Best Practices
Use functions

As well as loops and conditionals, Terraform functions are essential when it comes to achieving genericity in your code. They make your code more dynamic and ensure your configuration is DRY.

Functions allow you to perform various operations, such as converting expressions to different data types, calculating lengths, and building complex variables.

Terraform – Best Practices
Take advantage of Dynamic Blocks

Your code cannot be DRY without Dynamic Blocks. When this feature became available, it really helped with achieving the flexibility of building resources however you liked. Some cloud providers don't have dedicated resources for security group rules, for example, and those rules are usually embedded in the security group itself.

Without dynamic blocks, you would've always needed to change the configuration whenever a new rule was added, now you are simply changing the input.

Terraform – Best Practices

Use the Lifecycle Block

Sometimes, you may have some sophisticated conditions inside your code. Maybe you have a script that has to change something outside Terraform on your resources tags (this is not recommended, by the way, but it is just an example). You can use the Terraform lifecycle block to ignore changes on the tags, ensuring that you are not rolling back to the previous version.

The lifecycle block can also help if you have a pesky resource that, for some reason, seems like it is working properly, but you have to recreate it without downtime. You can use the `create_before_destroy=true` option for this.

Terraform – Best Practices
Use variables validations

Terraform does a pretty good job when it comes to validating that your variables are receiving the correct inputs, but what about when you want to restrict something and you don't have Policy as Code implemented?

Well, you can easily use variable validations to accommodate that. This validation block exists inside the variable, you can use terraform functions inside of it, and you can provide error messages if the condition is not respected.

Terraform – Best Practices

Leverage Helper tools to make your life easier

- tflint – Terraform linter for errors that the plan can't catch.
- tfenv – Terraform version manager (Read more about it in the How to Use tfenv to Manage Multiple Terraform Versions article)
- checkov – Terraform static analysis tool
- terratest – Go library that helps you with automated tests for Terraform
- pre-commit-terraform – Pre-commit git hooks for automation
- terraform-docs – Quickly generate docs from modules
- spacelift – Collaborative Infrastructure Delivery Platform for Terraform (Terraform Cloud alternative)
- atlantis – Workflow for collaborating on Terraform projects
- terraform-cost-estimation – Free cost estimation service for your plans.

Terraform – Best Practices
ake advantage of the IDE extensions

If you are using Visual Studio Code, or any other IDE when you are writing Terraform code, you can take advantage of their extensions to speed up your development process and also make sure your code is formatted correctly.

On vscode, you can use the Terraform extension built by Anton Kulikov. Keep in mind that you need to have Terraform installed on your local machine to ensure this is working properly.