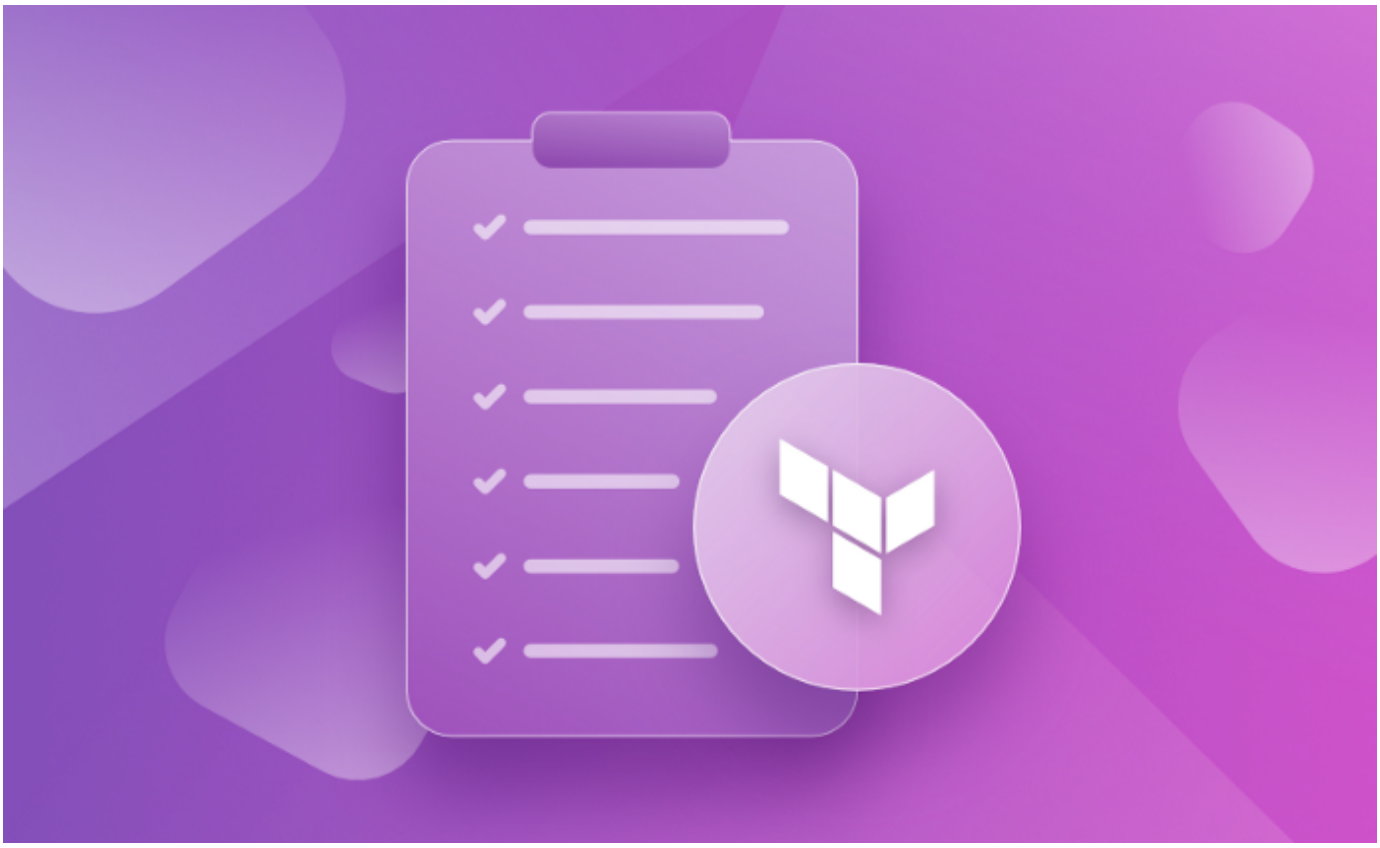
Terraform Best Practices for Better Infrastructure Management



In this article, we explore best practices for managing Infrastructure as Code (IaC) with Terraform. Terraform is one of the most used tools in the IaC space that enables us to safely and predictably apply changes to our infrastructure.

Starting with Terraform can feel like an intimidating task at first, but a beginner can quickly reach a basic understanding of the tool. After the initial learning period, a new user can then start running commands, creating, and refactoring Terraform code. During this process, many new users face nuances and issues around how to structure their code correctly, use advanced features, apply software development best practices in their IaC process, etc.

Let's check together some best practices that will assist you in pushing your Terraform skills to the next level. If you are entirely new to Terraform, look at the Terraform Spacelift Blog, where you can find a plethora of material, tutorials, and examples to boost your skills.

Terraform Key Concepts

In this section, we will describe some key Terraform concepts briefly. If you are already familiar with these, you can skip this section.

Terraform Configuration Language

Terraform uses its own configuration language to declare infrastructure objects and their associations. The goal of this language is to be declarative and describe the system's state that we want to reach.

Resources

Resources represent infrastructure objects and are one of the basic blocks of the Terraform language.

Data Sources

Data sources feed our Terraform configurations with external data or data defined by separate Terraform projects.

Modules

Modules help us group several resources and are the primary way to package resources in Terraform for reusability purposes.

State

Terraform keeps the information about the *state* of our infrastructure to store track mappings to our live infrastructure and metadata, create plans and apply new changes.

Providers

To interact with resources on cloud providers, Terraform uses some plugins named *providers*.

IaC Best Practices

Before moving to Terraform, first, let's check some fundamental best practices that apply to all Infrastructure as Code projects. These should be used in your processes regardless of your tool to manage your cloud infrastructure.

1. Use version control and prevent manual changes

This seems like an obvious statement in 2022, but it's the basis of everything else. We should treat our infrastructure configurations as application code and apply the same best practices for managing, testing, reviewing, and bringing it to production. We should embrace a GitOps approach that fits our use case and implement an automated CI/CD workflow for applying changes.

2. Shift your culture to Collaborative IaC

Keeping your infrastructure in version-controlled repositories is the first step to improving your manual infrastructure management. Next, we should strive to enable usage across teams with self-service infrastructure, apply policies and compliance according to our organization's standards, and access relevant insights and information. Thankfully, Spacelift can assist you along the way to achieving these.

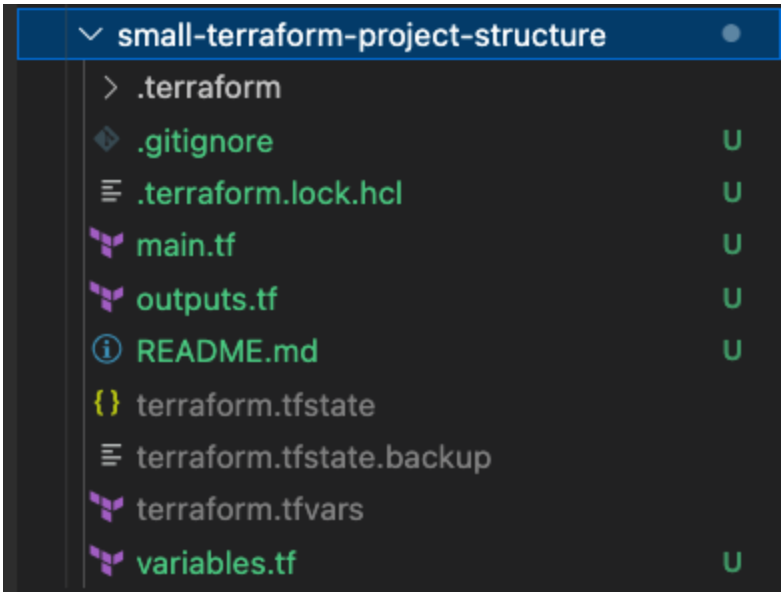
How to Structure Your Terraform Projects

This section will explore some strategies for structuring our Terraform projects. In the world of Terraform, there is no right or wrong way to structure our configurations, and most of the suggested structures that you will find online are heavily opinionated.

When deciding how to set up your Terraform configuration, the most important thing is to **understand your infrastructure needs** and your use case and **craft a solution that fits your team and the project**.

If we are dealing with a small project with limited infrastructure components, it's not a bad idea to keep our Terraform configuration as simple as possible. In these cases, we can configure only the necessary files for our *root module*, which are the configuration files that exist in the root directory. A small project can contain just these files **main.tf**, **variables.tf**, **README.md**. Some other files that you might find handy to use are the **outputs.tf** to define the output values of your project, **versions.tf** to gather any pinned versions for the configurations, and **providers.tf** to configure options related to the providers you use, especially if there are multiple.

Our primary entry point is *main.tf,* and in simple use cases, we can add all our resources there. We define our variables in *variables.tf* and assign values to them in *terraform.tfvars.* We use the file *outputs.tf* to declare output values. You can find a similar example project structure here.



When dealing with larger projects, things get a bit more complicated, and we have to take a step back to figure out the best structure for our projects.

We first have to **break down our Terraform code into reusable components** that abstract details from consumers and different teams can use and customize accordingly. We can achieve this by creating separate *modules* for pieces of infrastructure that should be reused in different environments, projects, and teams.

A common practice is to separate our modules according to ownership and responsibility, rate of change, and ease of management. For every module, we need to define its inputs and outputs and document them thoroughly to enable consumers to use them effectively. We can then leverage *outputs* and *terraform_remote_state* to reference values across modules or even different Terraform states. Beware that using the *terraform_remote_state* data source implies access to the entire state snapshot and this might post a security issue. Another option to share parameters between different states is to leverage an external tool for publishing and consuming the data like Amazon SSM Parameter Store or HashiCorp Consul.

We have to make the next decision to either keep all our Terraform code in a single repository (*monorepo*) or to separate our Terraform configurations in multiple code repositories. This is considered a great debate since both approaches have drawbacks and benefits. There is a tendency in the industry to avoid gigantic monorepos and use separate configurations to enable faster module development and flexibility. Personally, this is the approach I prefer too.

Usually, we have to deal with a plethora of different Infrastructure environments, and there are multiple ways to handle this in Terraform. A good and easy practice to follow is to have **separate Terraform configurations for different environments**. This way, different environments have their own state and can be tested and managed separately, while shared behavior is achieved with shared or remote modules.

One option is using a separate directory per environment and keeping a separate state for each directory. Another option would be to keep all the Terraform configurations in the same directory and pass different environment variables per environment to parametrize the configuration accordingly. Check out the Evolving Your Infrastructure with Terraform and How I Manage More Environments with Less Code in Terraform talks for some inspiration around structuring your projects.

Here, you can find an example structure for three different environments per directory: *production*, *staging*, and *test.* Each environment has its own state and is managed separately from the others while leveraging common or shared modules. Although this approach comes with some code duplication, we gain improved clarity, environment isolation, and scalability.

As a general rule, we want to define Terraform configurations with **limited scope** and blast radius with specific owners. To minimize risk, we should try to decompose our projects into small workspaces/stacks and segment access to them using Role-based access control(RBAC).

Terraform Specific Best Practices

Alright, in the previous sections, we talked about some generic IaC best practices. We explored some options for optimizing our Terraform code according to our organizational structure and needs.

In this part, we are deep-diving into specific points that will take our Terraform code to the next level. This list isn't exhaustive, and some parts are opinionated based on my personal preferences and experiences.

The goal here is to give you hints and guidance on experimenting, researching, and implementing the practices that make sense to your use case.

1. 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.

2. 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.

3. 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.

4. 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”
```

We can get it from a data source:

```
data "aws_caller_identity" "current" {}

locals {
    account_id    = data.aws_caller_identity.current.account_id
}
```

5. 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.

6. 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.*

7. 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* arguments should be defined after *tags* if relevant). A handy option to help you with tagging is defining some *default_tags* that apply to all resources managed by a provider. Check out this example to see how to set and override default tags for the AWS provider. If the provider you use doesn't support default tags, you must manually pass these tags through to your modules and apply them to your resources.

8. Introduce Policy as Code

As our teams and infrastructure scale, our trust in individual users is generally reduced. We should set up some policies to ensure our systems continue to be operational and secure. Having a *Policy as Code* process in place allows us to define the rules of what is considered secure and acceptable at scale and automatically verify these rules. Spacelift leverages an open-source engine to achieve this, Open Policy Agent(OPA).

9. 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**.

A more mature solution would be to set up a secret store like Hashicorp Vault or AWS Secrets Manager to handle access to secrets for you. This way, you can protect your secrets at rest and enforce encryption without too much trouble. You can also opt for more advanced features like secret rotation and audit logs. Beware that this approach usually comes with the cost of using this managed service.

10. 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.

There are a lot of tools out there that can help you with testing your Terraform code. I will list some of them in the "Helper Tools" section below.

11. 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* environment variable. Check out this Terraform Debug & Troubleshoot tutorial for some hands-on examples.

12. Leverage Helper tools to make your life easier

Terraform is one of the most used and loved IaC tools out there, and naturally, there is a big community around it. Along with this community, many helper tools are being constantly created to help us through our Terraform journey. Picking up and adopting the right tools for our workflows isn't always straightforward and usually involves an experimentation phase. Here you can find a list of tools that I find helpful based on my experience, but it is definitely not an exhaustive list.

- tflint — Terraform linter for errors that the plan can't catch.
- tfenv — Terraform version manager
- 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
- atlantis — Workflow for collaborating on Terraform projects
- terraform-cost-estimation — Free cost estimation service for your plans.

Check out also this list that includes a lot more awesome terraform tools.

Key Points

We have explored many different best practices for Terraform and Infrastructure as Code, analyzed various options for handling and structuring our Terraform projects, and saw how adopting helper tools could make our life easier.

Remember, this isn't a recipe that you have to follow blindly but a guide that aims to provide pointers and cues and trigger you to build your own optimal Terraform workflows and projects.

Thank you for reading, and I hope you enjoyed this "Terraform Best Practices" article as much as I did.