18 Most Useful Terraform Tools to Use in 2023

By itself, Terraform is an incredibly formidable Infrastructure-as-Code tool, but as you begin to add complexity to your infrastructure, you may find that it occasionally needs a helping hand. Let’s look at some of the most popular tools used in Terraform-managed deployments today.

# Native Tool

## 1. [Terraform Console](https://developer.hashicorp.com/terraform/cli/commands/console)

The Terraform Console is probably one of the most overlooked tools in the arsenal. This is surprising because it’s native to Terraform and is packaged within the binary!

The [terraform console](https://spacelift.io/blog/terraform-console) command provides an interactive console that can be used to test and evaluate any Terraform expressions you may wish to use in your code. Instead of endlessly experimenting, writing awkward outputs, and banging your head against the desk, the Terraform Console allows you to test out complicated expressions before you run your code.

Let’s take a quick look at an example:

$ terraform console  
Terraform console version 0.15.1  
  
> 10 + 20  
30  
  
> "hello, world"  
"hello, world"  
  
> length("hello, world")  
12  
  
> substr("hello, world", 0, 5)  
"hello"  
  
> exit

As you can see above, it’s useful for experimenting with expressions before you add them to your code. You can also evaluate variables and manipulate anything else in the state that you need.

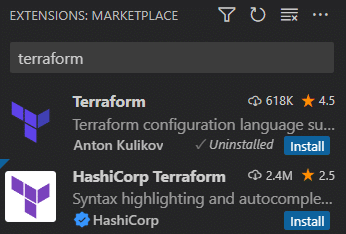
One thing to note is that the configuration must be able to pass a plan before the terraform console command will work, so make sure you comment out any issues you’re trying to solve before you enter the console to troubleshoot.

# IDE Tool

## 2. [Visual Studio Code](https://code.visualstudio.com/)

I won’t be listing more than this. VS Code has just absolutely eclipsed other IDEs in the development world. There are certainly some great IDEs out there, but the integrations and plugins in VS Code are unparalleled.

VS Code has two very popular Terraform extensions:



One is the official HashiCorp Terraform extension. It has more installations but also some pretty iffy reviews. It has seemed pretty stable whenever I’ve used it, but I typically default to the Terraform extension by Anton Kulikov. It has been solid and works very well. Your organization may have some policies around which extensions should be installed, so you should be good with either if they’re allowed.

# Linting Tool

# 3. [TFLint](https://github.com/terraform-linters/tflint" \t "_blank)

Tflint is an incredibly handy Terraform framework that lets you lint your Terraform code based on a prewritten ruleset or your own custom rules. These rules are added using “plugins.”

Some of the most common issues you may want to uncover using TFLint, as cited from [Github](https://github.com/terraform-linters/tflint" \t "_blank), are:

* Find possible errors, such as invalid instance types, for the major cloud providers, including AWS, GCP, and Azure.
* Warn about deprecated syntax and unused declarations.
* Enforce best practices and naming conventions.

An example is below.

# example.tf  
provider "aws" {  
 version = "2.70.0"  
 region = "us-west-2"  
}  
  
resource "aws\_instance" "example" {  
 ami = "ami-0c55b159cbfafe1f0"  
 instance\_type = "t2.micro"  
}

$ tflint  
  
example.tf  
 on main.tf line 4:  
 4: version = "2.70.0"  
 | ^ Error: AWS provider version should be specified in the required format: X.Y.Z (e.g. 2.1.0)  
  
Summary:  
 Count: 1  
 Invalid: 1  
 Warnings: 0  
 Errors: 1

# Security Tools

## 4. [Open Policy Agent](https://www.openpolicyagent.org/)

Open Policy Agent isn’t specifically a Terraform framework or tool, but it can be used directly with Terraform. OPA, as it is commonly called, is written in the [Rego](https://spacelift.io/blog/open-policy-agent-rego" \t "_blank) language, which is inspired by the old [Datalog](https://en.wikipedia.org/wiki/Datalog" \t "_blank) language. It is also used by several tools as their policy language when analyzing Terraform code.

Tools such as Terrascan, Spacelift, Terraform Cloud, and more all support it and use it to varying extents. Spacelift, for instance, uses it for every aspect of the experience. From the time a user logs in to Spacelift to the point where they’re launching stacks, OPA Policies are there, ensuring everything goes according to plan.

Here is an example of a policy that prevents Terraform from deploying AWS IAM access keys:

package spacelift  
  
# Note that the message here is dynamic and captures resource address to provide  
# appropriate context to anyone affected by this policy. For the sake of your  
# sanity and that of your colleagues, please a  
#  
# You can read more about plan policies here:  
# https://docs.spacelift.io/concepts/policy/terraform-plan-policy  
  
deny[sprintf(message, [resource.address])] {  
 message := "Static AWS credentials are evil (%s)"  
  
 resource := input.terraform.resource\_changes[\_]  
 resource.change.actions[\_] == "create"  
  
 # This is what decides whether the rule captures a resource.  
 # There may be an arbitrary number of conditions, and they all must  
 # succeed for the rule to take effect.  
 resource.type == "aws\_iam\_access\_key"  
}  
  
# Learn more about sampling policy evaluations here:  
# https://docs.spacelift.io/concepts/policy#sampling-policy-inputs  
sample = true

## 5. [Terrascan](https://runterrascan.io/docs/" \t "_blank)

Although the name “Terrascan” leads one to believe that it’s Terraform-specific, it actually works for many IaC providers. Terraform, Kubernetes, Ansible, Cloudformation, and more are supported by Terrascan thanks to its large set of pre-written [policies](https://runterrascan.io/docs/policies/). Of course, you can write your own policies as well for custom use cases.

Terrascan integrates with [several CI/CD](https://runterrascan.io/docs/integrations/) tools and is extremely flexible when designing your pipelines.

Here is a basic Terrascan usage example:

$ terrascan scan -t aws  
results:  
 violations:  
 - rule\_name: scanOnPushDisabled  
 description: Unscanned images may contain vulnerabilities  
 rule\_id: AWS.ECR.DataSecurity.High.0578  
 severity: MEDIUM  
 category: Data Security  
 resource\_name: scanOnPushDisabled  
 resource\_type: aws\_ecr\_repository  
 file: ecr.tf  
 line: 1  
 count:  
 low: 0  
 medium: 1  
 high: 0  
 total: 1

## 6. [Checkov](https://www.checkov.io/" \t "_blank)

Checkov is very similar to Terrascan, but it uses a [Python policy-as-code](https://www.checkov.io/3.Custom%20Policies/Python%20Custom%20Policies.html) framework instead of the Rego syntax found in OPA. Many engineers may find this more approachable as Python is one of the most popular programming languages today.

Although Python is more approachable for many, the syntax is still a little wordy and can get a little complex, as shown from the official docs below:

from lark import Token  
  
from checkov.terraform.checks.resource.base\_resource\_check import BaseResourceCheck  
from checkov.common.models.enums import CheckResult, CheckCategories  
  
class S3PCIPrivateACL(BaseResourceCheck):  
 def \_\_init\_\_(self):  
 name = "Ensure PCI Scope buckets has private ACL (enable public ACL for non-pci buckets)"  
 id = "CKV\_AWS\_999"  
 supported\_resources = ['aws\_s3\_bucket']  
 # CheckCategories are defined in models/enums.py  
 categories = [CheckCategories.BACKUP\_AND\_RECOVERY]  
 guideline = "Follow the link to get more info https://docs.bridgecrew.io/docs"  
 super().\_\_init\_\_(name=name, id=id, categories=categories, supported\_resources=supported\_resources, guideline=guideline)  
   
 def scan\_resource\_conf(self, conf):  
 """  
 Looks for ACL configuration at aws\_s3\_bucket and Tag values:  
 https://www.terraform.io/docs/providers/aws/r/s3\_bucket.html  
 :param conf: aws\_s3\_bucket configuration  
 :return: <CheckResult>  
 """  
 if 'tags' in conf.keys():  
 environment\_tag = Token("IDENTIFIER", "Scope")  
 if environment\_tag in conf['tags'][0].keys():  
 if conf['tags'][0][environment\_tag] == "PCI":  
 if 'acl' in conf.keys():  
 acl\_block = conf['acl']  
 if acl\_block in [["public-read"], ["public-read-write"], ["website"]]:  
 return CheckResult.FAILED  
 return CheckResult.PASSED  
  
check = S3PCIPrivateACL()

The “wordiness” may be completely subjective. Python is an excellent language, and I’m sure people would happily debate Rego vs. Python for hours on end.

Overall, the differences between Terrascan and Checkov come down to the policy language, as most other features are at parity with each other.

## 7. [Tfsec](https://github.com/aquasecurity/tfsec" \t "_blank)

The tfsec tool is another Rego-based policy tool that is very similar to Terrascan and Checkov. One of the standout features is its multiple policy creation formats. You can use JSON, YAML, or Rego. Rego will be the most flexible and powerful, but JSON and YAML can create most policies you would need with a much more readable format.

Here is an example of a check file in YAML:

checks:  
- code: CUS001  
 description: Custom check to ensure the CostCentre tag is applied to EC2 instances  
 impact: By not having CostCentre we can't keep track of billing  
 resolution: Add the CostCentre tag  
 requiredTypes:  
 - resource  
 requiredLabels:  
 - aws\_instance  
 severity: ERROR  
 matchSpec:  
 name: tags  
 action: contains  
 value: CostCentre  
 errorMessage: The required CostCentre tag was missing  
 relatedLinks:  
 - http://internal.acmecorp.com/standards/aws/tagging.html

# Drift Tool

## 8. [Driftctl](https://driftctl.com/" \t "_blank)

Driftctl allows you to perform one of the most crucial aspects of IaC management, which is to detect resources that have drifted from your desired state. It is capable of finding managed resources that have drifted as well as unmanaged resources that were created outside of driftctl.

An example output is here:

Found missing resources:  
 aws\_s3\_bucket:  
 - driftctl-bucket-test-2  
Found resources not covered by IaC:  
 aws\_s3\_bucket:  
 - driftctl-bucket-test-3  
Found changed resources:  
 - driftctl-bucket-test-1 (aws\_s3\_bucket):  
 ~ Versioning.0.Enabled: false => true  
Found 3 resource(s)  
 - 33% coverage  
 - 1 covered by IaC  
 - 1 not covered by IaC  
 - 1 missing on cloud provider  
 - 1/1 changed outside of IaC

# Costing Tool

## 9. [Infracost](https://www.infracost.io/" \t "_blank)

Infracost is probably one of the most popular tools used with Terraform in production deployments. It queries the pricing API of the provider and displays a cost based on what you plan to deploy.

The great thing isn’t only that it shows you what it’s going to cost, but you can use a policy tool, such as Open Policy Agent, to write policies that can block a deployment based on the cost. CI/CD tools, such as Spacelift, can easily add tools like this into the deployment pipeline with very minimal configuration. Talk about a CFO’s dream! The community version is completely free but will require some custom tooling to get functionality like the policies mentioned above. If you choose the paid version, all of that is included. There is even a [VS Code](https://github.com/infracost/vscode-infracost/) extension to make things even more streamlined.

Here is an example of the truncated output:

Project: my-terraform-project  
Region: us-west-2  
  
+ aws\_ebs\_volume.my\_volume  
 Type: aws\_ebs\_volume  
 Quantity: 1  
 Monthly cost: $10.50  
  
+ aws\_instance.my\_instance  
 Type: aws\_instance  
 Quantity: 1  
 Monthly cost: $100.00  
  
Total monthly cost: $110.50

Read more about [InfraCost and how to estimate cloud costs with Terraform](https://spacelift.io/blog/terraform-cost-estimation-using-infracost" \t "_blank).

# Terraform HCL Generation Tools

## 10. [Terraformer](https://github.com/GoogleCloudPlatform/terraformer" \t "_blank)

Terraformer is an extremely cool product that scans your cloud environment and outputs the necessary Terraform code to redeploy it. This is definitely not the preferred route to codify your resources as, by now, most startups should have started with Terraform on day one. Unfortunately, many companies started before Terraform was the great tool it is today, so tools like Terraformer are ideal for getting things under new management.

Terraformer is great in a lot of ways, but you’ll want to be extremely careful how you import resources and that you truly understand the infrastructure that it creates and how you structure it. These types of tools rarely create perfect code at best and create an absolute mess of unreadable code at worst. I would strongly recommend reading ALL of the code that’s created and performing a thorough audit to ensure everything is in a readable and scalable structure.

## 11. [Pike](https://github.com/JamesWoolfenden/pike)

Pike is an interesting tool that will analyze the resources you wish to create using Terraform and generate the necessary IAM permissions you need to complete that deployment.

An example of this is here:

./pike scan -d .\terraform\  
{  
 "Version": "2012-10-17",  
 "Statement": {  
 "Effect": "Allow",  
 "Action": [  
 "ec2:MonitorInstances",  
 "ec2:UnmonitorInstances",  
 "ec2:DescribeInstances",  
 "ec2:DescribeTags",  
 "ec2:DescribeInstanceAttribute",  
 "ec2:DescribeVolumes",  
 "ec2:DescribeInstanceTypes",  
 "ec2:RunInstances",  
 "ec2:DescribeInstanceCreditSpecifications",  
 "ec2:StopInstances",  
 "ec2:StartInstances",  
 "ec2:ModifyInstanceAttribute",  
 "ec2:TerminateInstances",  
 "ec2:AuthorizeSecurityGroupIngress",  
 "ec2:AuthorizeSecurityGroupEgress",  
 "ec2:CreateSecurityGroup",  
 "ec2:DescribeSecurityGroups",  
 "ec2:DescribeAccountAttributes",  
 "ec2:DescribeNetworkInterfaces",  
 "ec2:DeleteSecurityGroup",  
 "ec2:RevokeSecurityGroupEgress"  
 ],  
 "Resource": "\*"  
 }  
}

Like any security-related tool, ensure you audit the permissions afterward to ensure compliance.

# Documentation Generation Tool

## 12. [Terraform Docs](https://github.com/terraform-docs/terraform-docs)

Let’s face it, writing documentation isn’t every developer’s dream job. Luckily, there’s an excellent tool to automatically generate your Terraform docs from configuration files. Terraform-docs is extremely useful and simple to use when creating documentation that shows exactly what your users need to know. It exposes variables you can add to a template file, and it will automatically generate the sections for you in an easy-to-read format.

You can find an example of a repository that uses this [here](https://github.com/flavius-dinu/terraform-az-aks).

And an example of the format is below:

content: |-  
 Any arbitrary text can be placed anywhere in the content  
  
 {{ .Header }}  
   
 and even in between sections  
   
 {{ .Providers }}  
   
 and they don't even need to be in the default order  
   
 {{ .Outputs }}  
  
 include any relative files  
  
 {{ include "relative/path/to/file" }}  
   
 {{ .Inputs }}  
   
 # Examples  
  
 ```hcl  
 {{ include "examples/foo/main.tf" }}  
 ```  
  
 ## Resources  
   
 {{ range .Module.Resources }}  
 - {{ .GetMode }}.{{ .Spec }} ({{ .Position.Filename }}#{{ .Position.Line }})  
 {{- end }}

# Management Tools

## 13. [TFSwitch](https://tfswitch.warrensbox.com/" \t "_blank)

TFSwitch is a simple utility that helps to manage multiple Terraform versions. You can switch between any available Terraform version. The tool will also download the version you need if required. When working with multiple Terraform code bases, this is absolutely critical to keep pesky syntax errors and other headaches at bay if you’re dealing with code that has deprecated or newer features than your current version.

To use TFSwitch, you just need to install it using the instructions [here](https://tfswitch.warrensbox.com/Install/), and run the tfswitch command to display all of the available versions:

tfswitch  
Creating directory for terraform binary at: /home/user/.terraform.versions  
Use the arrow keys to navigate: ↓ ↑ → ←  
? Select Terraform version:  
 ▸ 1.3.9  
 1.3.8  
 1.3.7  
 1.3.6  
↓ 1.3.5

Once you choose the version you need, you are free to use Terraform!

## 14. [Terramate](https://github.com/mineiros-io/terramate" \t "_blank)

Terramate is an interesting tool that can help you manage large Terraform deployments by breaking the code into “stacks.” These stacks have separate states and allow your code to be deployed with a smaller blast radius. It also has features that work with git to deploy automatically on changes. Where Terramate really shines is its ability to keep your code DRY (Do not Repeat Yourself). It is able to share variables, provider configurations, and other data between stacks and automatically generate any code necessary in pure Terraform.

The tool is fairly complex in all of its features, so it’s probably best to read the [release blog](https://blog.mineiros.io/introducing-terramate-an-orchestrator-and-code-generator-for-terraform-5e538c9ee055) to let you dive in.

## 15. [Terragrunt](https://terragrunt.gruntwork.io/" \t "_blank)

Terragrunt is a very popular Terraform tool that, like Terramate, helps keep your [Terraform code DRY](https://spacelift.io/blog/terragrunt-with-spacelift), manages multiple stacks, and more. It makes it easier to work with multiple cloud accounts, modules, and more with features that greatly increase the efficiency of your code. Terraform’s latest versions have honestly caught up with a lot of the features in Terragrunt, but definitely not all.

Here is a sample configuration file.

# Indicate where to source the terraform module from.  
# The URL used here is a shorthand for  
# "tfr://registry.terraform.io/terraform-aws-modules/vpc/aws?version=3.5.0".  
# Note the extra `/` after the protocol is required for the shorthand  
# notation.  
terraform {  
 source = "tfr:///terraform-aws-modules/vpc/aws?version=3.5.0"  
}  
  
# Indicate what region to deploy the resources into  
generate "provider" {  
 path = "provider.tf"  
 if\_exists = "overwrite\_terragrunt"  
 contents = <<EOF  
provider "aws" {  
 region = "us-east-1"  
}  
EOF  
}  
  
# Indicate the input values to use for the variables of the module.  
inputs = {  
 name = "my-vpc"  
 cidr = "10.0.0.0/16"  
 azs = ["us-east-1a", "us-east-1b", "us-east-1c"]  
 private\_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]  
 public\_subnets = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]  
 enable\_nat\_gateway = true  
 enable\_vpn\_gateway = false  
 tags = {  
 Terraform = "true"  
 Environment = "dev"  
 }  
}

As you can see in the comments, it makes it very easy to generate provider blocks for all modules and inject variables where they’re needed without having to pass them manually all over the place. It’s definitely an interesting project and something worth checking out if you start to juggle a lot of modules.

Check out our [Terragrunt vs. Terraform comparison](https://spacelift.io/blog/terragrunt-vs-terraform" \t "_blank) to learn more.

# CI/CD Tools

## 16. [Atlantis](https://www.runatlantis.io/)

Atlantis is a great open-source project for Terraform pull-request automation. While that sounds a little niche, it’s actually incredibly important to nail down your GitOps workflow. When dealing with multiple developers, committing code, creating a pull request, and having a platform inspect that pull request and deploy your code is critical to maintaining deployment velocity.

Within Atlantis, you can also integrate other tools listed in this article to further enrich your pipeline and add functionality and guardrails to ensure reliable deployments. Atlantis is fairly opinionated about its pull request workflows, but if the workflow fits your GitOps strategy, it’s an excellent open-source option if you use Terraform for your IaC.

Check out a great comparison of [Spacelift and Atlantis](https://spacelift.io/blog/alternative-to-atlantis" \t "_blank).

## 17. [Terraform Cloud](https://cloud.hashicorp.com/products/terraform)

Terraform Cloud is part of the Hashicorp Cloud Platform (HCP) that, like Atlantis, manages Terraform deployments triggered by commits to your VCS.

Terraform Cloud has excellent state management features, and the deployment pipeline can be customized with most of the tools featured in this article. Unfortunately, the ability to integrate external applications currently requires a paid plan. Policy as code with Hashicorp’s proprietary Sentinel engine or with OPA is also possible but requires a paid plan as well.

Terraform Cloud manages your variables, secrets, resources, and more in its offering. It also allows you to script itself in Terraform, so everything can be automated. While it only supports Terraform, it does so very well. The free version is decent for a small team, but [pricing escalates once you scale](https://spacelift.io/blog/terraform-cloud-pricing).

## 18. [Spacelift](https://spacelift.io/" \t "_blank)

Spacelift shares a lot of features with Atlantis, Terraform Cloud, and other “Terraform Automation and Collaboration Software” (TACOS), but keeps developers, policies, and simplicity in mind from the moment you sign in. You can sign into Spacelift using SSO from your VCS provider and [get going right away](https://docs.spacelift.io/getting-started). Spacelift support Terraform as well as Cloudformation, Ansible, Pulumi, and Kubernetes, which sets it apart from other CI/CD tools mentioned in this list.

Spacelift is built with policies in mind and allows [OPA policies](https://docs.spacelift.io/concepts/policy/) to be written that can manage nearly every aspect of the platform. From login, stack access, planning, deployment, and more. You can also [integrate](https://docs.spacelift.io/concepts/stack/stack-settings#customizing-workflow) any of the tools mentioned in the other sections of this article, such as Infracost and Terrascan, to check your code before deployment (see more examples: [Integrating Security Tools With Spacelift](https://spacelift.io/blog/integrating-security-tools-with-spacelift)). Writing policies based on the findings of these tools is also a really great feature that allows you to manage every aspect of your IaC strategy.

Spacelift also manages state, secrets, [cloud credentials](https://docs.spacelift.io/integrations/cloud-providers/), and more through an intuitive user interface or by scripting everything in Terraform. If you have multiple stacks that need to be run, you can chain those together using stack dependencies. This is helpful if you need to run an Ansible playbook on newly created resources.

Spacelift has a generous free tier for a single user that covers most of the features. Like Terraform Cloud, you’ll need to upgrade to get more advanced features, such as private workers, stack scheduling, and an incredibly useful [Drift Detection](https://spacelift.io/drift-detection) feature.

# Key Points

That’s a quick run-through of several popular Terraform tools you’ll find in the wild today. New tools come out daily, and the popularity of these tools ebbs and flows frequently. Luckily, most of these tools work in very similar ways and make it very easy to move between them.

Make sure you check them out and especially give Spacelift a go for a fully-featured [free trial](https://spacelift.io/free-trial)!