at test time from several hundred to just a handful so that these tests will be faster and less brittle.

Moreover, this approach to end-to-end testing more closely mimics how you'll be deploying those changes in production. After all, it's not like you tear down and bring up your production environment from scratch to roll out each change. Instead, you apply each change incrementally, so this style of end-to-end testing offers a huge advantage: you can test not only that your infrastructure works correctly but also that the *deployment process* for that infrastructure works correctly, too.

Other Testing Approaches

Most of this chapter has focused on testing your Terraform code by doing a full `apply` and `destroy` cycle. This is the gold standard of testing, but there are three other types of automated tests you can use:

- Static analysis
- Plan testing
- Server testing

Just as unit, integration, and end-to-end tests each catch different types of bugs, each of the testing approaches just mentioned will catch different types of bugs as well, so you'll most likely want to use several of these techniques together to get the best results. Let's go through these new categories one at a time.

Static analysis

Static analysis is the most basic way to test your Terraform code: you parse the code and analyze it without actually executing it in any way. Table 9-1 shows some of the tools in this space that work with Terraform and how they compare in terms of popularity and maturity, based on stats I gathered from GitHub in February 2022.

Table 9-1. A comparison of popular static analysis tools for Terraform

	`terraform validate`	`tfsec`	`tflint`	**Terrascan**
Brief description	Built-in Terraform command	Spot potential security issues	Pluggable Terraform linter	Detect complia and security violations
License	(same as Terraform)	MIT	MPL 2.0	Apache 2.0
Backing company	(same as Terraform)	Aqua Security	(none)	Accurics
Stars	(same as Terraform)	3,874	2,853	2,768
Contributors	(same as Terraform)	96	77	63
First release	(same as Terraform)	2019	2016	2017
Latest release	(same as Terraform)	v1.1.2	v0.34.1	v1.13.0
Built-in checks	Syntax checks only	AWS, Azure, GCP, Kubernetes, DigitalOcean, etc.	AWS, Azure, and GCP	AWS, Azure, C Kubernetes, etc
Custom checks	Not supported	Defined in YAML or JSON	Defined in a Go plugin	Defined in Reg

The simplest of these tools is `terraform validate`, which is built into Terraform itself, which can catch syntax issues. For example, if you forgot to set the `alb_name` parameter in *examples/alb*, and you ran `validate`, you would get output similar to the following:

```
$ terraform validate

│ Error: Missing required argument
```

```
│
│   on main.tf line 20, in module "alb":
│   20: module "alb" {
│
│ The argument "alb_name" is required, but no definition was
found.
```

Note that `validate` is limited solely to syntactic checks, whereas the other tools allow you to enforce other types of policies. For example, you can use tools such as `tfsec` and `tflint` to enforce policies, such as:

- Security groups cannot be too open: e.g., block inbound rules that allow access from all IPs (CIDR block `0.0.0.0/0`).
- All EC2 Instances must follow a specific tagging convention.

The idea here is to *define your policies as code*, so you can enforce your security, compliance, and reliability requirements as code. In the next few sections, you'll see several other policy as code tools.

Strengths of static analysis tools

- They run fast.
- They are easy to use.
- They are stable (no flaky tests).
- You don't need to authenticate to a real provider (e.g., to a real AWS account).
- You don't have to deploy/undeploy real resources.

Weaknesses of static analysis tools

- They are very limited in the types of errors they can catch. Namely, they can only catch errors that can be determined from statically reading the code, without executing it: e.g., syntax errors, type errors, and a small subset of business logic errors. For example, you can detect a policy violation for static values, such as a security