group hardcoded to allow inbound access from CIDR block `0.0.0.0/0`, but you can't detect policy violations from dynamic values, such as the same security group but with the inbound CIDR block being read in from a variable or file.

- These tests aren't checking functionality, so it's possible for all the checks to pass and the infrastructure still doesn't work!

Plan testing

Another way to test your code is to run `terraform plan` and to analyze the plan output. Since you're executing the code, this is more than static analysis, but it's less than a unit or integration test, as you're not executing the code fully: in particular, `plan` executes the read steps (e.g., fetching state, executing data sources) but not the write steps (e.g., creating or modifying resources). Table 9-2 shows some of the tools that do `plan` testing and how they compare in terms of popularity and maturity, based on stats I gathered from GitHub in February 2022.

Table 9-2. A comparison of popular plan testing tools for Terraform

	Terratest	Open Policy Agent (OPA)	HashiCorp Sentinel	Checkov
Brief description	Go library for IaC testing	General-purpose policy engine	Policy-as-code for HashiCorp enterprise products	Policy-as-code everyone
License	Apache 2.0	Apache 2.0	Commercial / proprietary license	Apache 2.0
Backing company	Gruntwork	Styra	HashiCorp	Bridgecrew
Stars	5,888	6,207	(not open source)	3,758
Contributors	157	237	(not open source)	199
First release	2016	2016	2017	2019
Latest release	v0.40.0	v0.37.1	v0.18.5	2.0.810
Built-in checks	None	None	None	AWS, Azure, C Kubernetes, etc
Custom checks	Defined in Go	Defined in Rego	Defined in Sentinel	Defined in Pytl or YAML

Since you're already familiar with Terratest, let's take a quick look at how you can use it to do `plan` testing on the code in *examples/alb*. If you ran `terraform plan` manually, here's a snippet of the output you'd get:

```
Terraform will perform the following actions:

  # module.alb.aws_lb.example will be created
  + resource "aws_lb" "example" {
      + arn                        = (known after apply)
      + load_balancer_type         = "application"
      + name                       = "test-4Ti6CP"
      (...)
    }

  (...)
```

```
Plan: 5 to add, 0 to change, 0 to destroy.
```

How can you test this output programmatically? Here's the basic structure of a test that uses Terratest's `InitAndPlan` helper to run `init` and `plan` automatically:

```
func TestAlbExamplePlan(t *testing.T) {
        t.Parallel()

        albName := fmt.Sprintf("test-%s", random.UniqueId())

        opts := &terraform.Options{
                // You should update this relative path to point
at your alb
                // example directory!
                TerraformDir: "../examples/alb",
                Vars: map[string]interface{}{
                        "alb_name": albName,
                },
        }

        planString := terraform.InitAndPlan(t, opts)
}
```

Even this minimal test offers some value, in that it validates that your code can successfully run `plan`, which checks that the syntax is valid and that all the read API calls work. But you can go even further. One small improvement is to check that you get the expected counts at the end of the plan: "5 to add, 0 to change, 0 to destroy." You can do this using the `GetResourceCount` helper

```
        // An example of how to check the plan output's
add/change/destroy counts
        resourceCounts := terraform.GetResourceCount(t,
planString)
        require.Equal(t, 5, resourceCounts.Add)
        require.Equal(t, 0, resourceCounts.Change)
        require.Equal(t, 0, resourceCounts.Destroy)
```

You can do an even more thorough check by using the `InitAndPlanAndShowWithStructNoLogTempPlanFile` helper to parse the `plan` output into a `struct`, which gives you programmatic access to all the values and changes in that `plan` output. For example, you could check that the `plan` output includes the `aws_lb` resource at address `module.alb.aws_lb.example` and that the `name` attribute of this resource is set to the expected value, as follows:

```
        // An example of how to check specific values in the plan
output
        planStruct :=

terraform.InitAndPlanAndShowWithStructNoLogTempPlanFile(t, opts)

        alb, exists :=

planStruct.ResourcePlannedValuesMap["module.alb.aws_lb.example"]
        require.True(t, exists, "aws_lb resource must exist")

        name, exists := alb.AttributeValues["name"]
        require.True(t, exists, "missing name parameter")
        require.Equal(t, albName, name)
```

The strength of Terratest's approach to plan testing is that it's extremely flexible, as you can write arbitrary Go code to check whatever you want. But this very same factor is also, in some ways, a weakness, as it makes it harder to get started.

Some teams prefer a more declarative language for defining their policies as code. In the last few years, Open Policy Agent (OPA) has become a popular *policy-as-code* tool, as it allows your to capture you company's policies as code in a declarative language called Rego.

For example, many companies have tagging policies they want to enforce. A common one with Terraform code is to ensure that every resource that is managed by Terraform has a `ManagedBy = terraform` tag. Here is a simple policy called *enforce_tagging.rego* you could use to check for this tag:

```
package terraform

allow {
   resource_change := input.resource_changes[_]
   resource_change.change.after.tags["ManagedBy"]
}
```

This policy will look through the changes in a `terraform plan` output, extract the tag `ManagedBy`, and set an OPA variable called `allow` to `true` if that tag is set or `undefined` otherwise.

Now, consider the following Terraform module:

```
resource "aws_instance" "example" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
}
```

This module is not setting the required `ManagedBy` tag. How can we catch that with OPA?

The first step is to run `terraform plan` and to save the output to a plan file:

```
$ terraform plan -out tfplan.binary
```

OPA only operates on JSON, so the next step is to convert the plan file to JSON using the `terraform show` command:

```
$ terraform show -json tfplan.binary > tfplan.json
```

Finally, you can run the `opa eval` command to check this plan file against the *enforce_tagging.rego* policy:

```
$ opa eval \
  --data enforce_tagging.rego \
  --input tfplan.json \
  --format pretty \
  data.terraform.allow
```

```
undefined
```

Since the `ManagedBy` tag was not set, the output from OPA is `undefined`. Now, try setting the `ManagedBy` tag:

```
resource "aws_instance" "example" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

  tags = {
    ManagedBy = "terraform"
  }
}
```

Rerun `terraform plan`, `terraform show`, and `opa eval`:

```
$ terraform plan -out tfplan.binary

$ terraform show -json tfplan.binary > tfplan.json

$ opa eval \
  --data enforce_tagging.rego \
  --input tfplan.json \
  --format pretty \
  data.terraform.allow

true
```

This time, the output is `true`, which means the policy has passed.

Using tools like OPA, you can enforce your company's requirements by creating a library of such policies and setting up a CI/CD pipeline that runs these policies against your Terraform modules after every commit.

Strengths of plan testing tools

- They run fast—not quite as fast as pure static analysis but much faster than unit or integration tests.
- They are somewhat easy to use—not quite as easy as pure static analysis but much easier than unit or integration tests.