- They are stable (few flaky tests)—not quite as stable as pure static analysis but much more stable than unit or integration tests.
- You don't have to deploy/undeploy real resources.

Weaknesses of plan testing tools

- They are limited in the types of errors they can catch. They can catch more than pure static analysis but nowhere near as many errors as unit and integration testing.
- You have to authenticate to a real provider (e.g., to a real AWS account). This is required for `plan` to work.
- These tests aren't checking functionality, so it's possible for all the checks to pass and the infrastructure still doesn't work!

Server testing

There are a set of testing tools that are focused on testing that your servers (including virtual servers) have been properly configured. I'm not aware of any common name for these sorts of tools, so I'll call it *server testing*. These are not general-purpose tools for testing all aspects of your Terraform code. In fact, most of these tools were originally built to be used with configuration management tools, such as Chef and Puppet, which were entirely focused on launching servers. However, as Terraform has grown in popularity, it's now very common to use it to launch servers, and these tools can be helpful for validating that the servers you launched are working. Table 9-3 shows some of the tools that do server testing and how they compare in terms of popularity and maturity, based on stats I gathered from GitHub in February 2022.

Table 9-3. A comparison of popular server testing tools

	InSpec	Serverspec	Goss
Brief description	Auditing and testing framework	RSpec tests for your servers	Quick and easy server testing/validation
License	Apache 2.0	MIT	Apache 2.0
Backing company	Chef	(none)	(none)
Stars	2,472	2,426	4,607
Contributors	279	128	89
First release	2016	2013	2015
Latest release	v4.52.9	v2.42.0	v0.3.16
Built-in checks	None	None	None
Custom checks	Defined in a Ruby-based DSL	Defined in a Ruby-based DSL	Defined in YAML

Most of these tools provide a simple *domain-specific language* (DSL) for checking that the servers you've deployed conform to some sort of specification. For example, if you were testing a Terraform module that deployed an EC2 Instance, you could use the following `inspec` code to validate that the Instance has proper permissions on specific files, has certain dependencies installed, and is listening on a specific port:

```
describe file('/etc/myapp.conf') do
  it { should exist }
  its('mode') { should cmp 0644 }
end

describe apache_conf do
  its('Listen') { should cmp 8080 }
end

describe port(8080) do
  it { should be_listening }
end
```

Strengths of server testing tools

- They make it easy to validate specific properties of servers. The DSLs these tools offer are much easier to use for common checks than doing it all from scratch.
- You can build up a library of policy checks. Because each individual check is quick to write, per the previous bullet point, these tools tend to be a good way to validate a checklist of requirements, especially around compliance (e.g., PCI compliance, HIPAA compliance, etc.).
- They can catch many types of errors. Since you actually have to run `apply` and you validate a real, running server, these types of tests catch far more types of errors than pure static analysis or plan testing.

Weaknesses of server testing tools

- They are not as fast. These tests only work on servers that are deployed, so you have to run the full `apply` (and perhaps `destroy`) cycle, which can take a long time.
- They are not as stable (some flaky tests). Since you have to run `apply` and wait for real servers to deploy, you will hit various intermittent issues and occasionally have flaky tests.
- You have to authenticate to a real provider (e.g., to a real AWS account). This is required for the `apply` to work to deploy the servers, plus, these server testing tools all require additional authentication methods—e.g., SSH—to connect to the servers you're testing.
- You have to deploy/undeploy real resources. This takes time and costs money.