Yevgeniy Brikman Follow
Co-founder of Gruntwork, Author of "Hello, Startup" and "Terraform: Up & Running"
Sep 26, 2016 · 11 min read

Why we use Terraform and not Chef, Puppet, Ansible, SaltStack, or CloudFormation

1. *Update: we took this blog post series, expanded it, and turned it into a book called Terraform: Up & Running!*

This is Part 1 of the Comprehensive Guide to Terraform series. In the intro to the series, we discussed why every company should be using infrastructure-as-code (IAC). In this post, we're going to discuss why we picked Terraform as our IAC tool of choice.

If you search the Internet for "infrastructure-as-code", it's pretty easy to come up with a list of the most popular tools:

- • Chef
  • Puppet
  • Ansible

- SaltStack
- CloudFormation
- Terraform

What's not easy is figuring out which one of these you should use. All of these tools can be used to manage infrastructure as code. All of them are open source, backed by large communities of contributors, and work with many different cloud providers (with the notable exception of CloudFormation, which is closed source and AWS-only). All of them offer enterprise support. All of them are well documented, both in terms of official documentation and community resources such as blog posts and StackOverflow questions. So how do you decide?

What makes this even harder is that most of the comparisons you find online between these tools do little more than list the general properties of each tool and make it sound like you could be equally successful with any of them. And while that's technically true, it's not helpful. It's a bit like telling a programming newbie that you could be equally successful building a website with PHP, C, or Assembly—a statement that's technically true, but one that omits a huge amount of information that would be incredibly useful in making a good decision.

In this post, we're going to dive into some very specific reasons for why we picked Terraform over the other IAC tools. As with all technology decisions, it's a question of trade-offs and priorities, and while your particular priorities may be different than ours, we hope that sharing our thought process will help you make your own decision. Here are the main trade-offs we considered:

- Configuration Management vs Orchestration
- Mutable Infrastructure vs Immutable Infrastructure
- Procedural vs Declarative
- Client/Server Architecture vs Client-Only Architecture

Configuration Management vs Orchestration

Chef, Puppet, Ansible, and SaltStack are all "configuration management" tools, which means they are designed to install and manage software on existing servers. CloudFormation and Terraform are “orchestration tools”, which means they are designed to provision the servers themselves, leaving the job of configuring those servers to other tools. These two categories are not mutually exclusive, as most configuration management tools can do some degree of provisioning and most orchestration tools can do some degree of configuration management. But the focus on configuration management or orchestration means that some of the tools are going to be a better fit for certain types of tasks.

In particular, we’ve found that if you use Docker or Packer, the vast majority of your configuration management needs are already taken care of. With Docker and Packer, you can create images (such as containers or virtual machine images) that have all the software your server needs already installed and configured (for more info on what makes Docker great, see here). Once you have such an image, all you need is a server to run it. And if all you need to do is provision a bunch of servers, then an orchestration tool like Terraform is typically going to be a better fit than a configuration management tool (here’s an example of how to use Terraform to deploy Docker on AWS).

Mutable Infrastructure vs Immutable Infrastructure

Configuration management tools such as Chef, Puppet, Ansible, and SaltStack typically default to a mutable infrastructure paradigm. For example, if you tell Chef to install a new version of OpenSSL, it’ll run the software update on your existing servers and the changes will happen in-place. Over time, as you apply more and more updates, each server builds up a unique history of changes. This often leads to a phenomenon known as *configuration drift*, where each server becomes slightly different than all the others, leading to subtle configuration bugs that are difficult to diagnose and nearly impossible to reproduce.

If you’re using an orchestration tool such as Terraform to deploy machine images created by Docker or Packer, then every “change” is actually a deployment of a new server (just like every “change” to a variable in functional programming actually returns a new variable). For example, to deploy a new version of OpenSSL, you would create a new image using Packer or Docker with the new version of OpenSSL already installed, deploy that image across a set of totally new servers, and then undeploy the old servers. This approach reduces the likelihood of configuration drift bugs, makes it easier to know exactly what software is running on a server, and allows you to trivially deploy any previous version of the software at any time. Of course, it's possible to force configuration management tools to do immutable deployments too, but it's not the idiomatic approach for those tools, whereas it's a natural way to use orchestration tools.

Procedural vs Declarative

Chef and Ansible encourage a procedural style where you write code that specifies, step-by-step, how to to achieve some desired end state. Terraform, CloudFormation, SaltStack, and Puppet all encourage a more declarative style where you write code that specifies your desired end state, and the IAC tool itself is responsible for figuring out how to achieve that state.

For example, let's say you wanted to deploy 10 servers ("EC2 Instances" in AWS lingo) to run v1 of an app. Here is a simplified example of an Ansible template that does this with a procedural approach:

```
- ec2:
    count: 10
    image: ami-v1
    instance_type: t2.micro
```

And here is a simplified example of a Terraform template that does the same thing using a declarative approach:

```
resource "aws_instance" "example" {
  count = 10
  ami = "ami-v1"
  instance_type = "t2.micro"
}
```

Now at the surface, these two approaches may look similar, and when you initially execute them with Ansible or Terraform, they will produce similar results. The interesting thing is what happens when you want to make a change.

For example, imagine traffic has gone up and you want to increase the number of servers to 15. With Ansible, the procedural code you wrote earlier is no longer useful; if you just updated the number of servers to 15 and reran that code, it would deploy 15 new servers, giving you 25 total! So instead, you have to be aware of what is already deployed and write a totally new procedural script to add the 5 new servers:

```
- ec2:
    count: 5
    image: ami-v1
    instance_type: t2.micro
```

With declarative code, since all you do is declare the end state you want, and Terraform figures out how to get to that end state, Terraform will also be aware of any state it created in the past. Therefore, to deploy 5 more servers, all you have to do is go back to the same Terraform template and update the count from 10 to 15:

```
resource "aws_instance" "example" {
  count = 15
  ami = "ami-v1"
  instance_type = "t2.micro"
}
```

If you executed this template, Terraform would realize it had already created 10 servers and therefore that all it needed to do was create 5 new servers. In fact, before running this template, you can use Terraform's "plan" command to preview what changes it would make:

```
> terraform plan

+ aws_instance.example.11
    ami:                      "ami-v1"
    instance_type:            "t2.micro"

+ aws_instance.example.12
    ami:                      "ami-v1"
    instance_type:            "t2.micro"

+ aws_instance.example.13
```

```
    ami:                        "ami-v1"
    instance_type:              "t2.micro"

+ aws_instance.example.14
    ami:                        "ami-v1"
    instance_type:              "t2.micro"

+ aws_instance.example.15
    ami:                        "ami-v1"
    instance_type:              "t2.micro"

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

Now what happens when you want to deploy v2 the service? With the procedural approach, both of your previous Ansible templates are again not useful, so you have to write yet another template to track down the 10 servers you deployed previous (or was it 15 now?) and carefully update each one to the new version. With the declarative approach of Terraform, you go back to the exact same template once again and simply change the ami version number to v2:

```
resource "aws_instance" "example" {
  count = 15
  ami = "ami-v2"
  instance_type = "t2.micro"
}
```

Obviously, the above examples are simplified. Ansible does allow you to use tags to search for existing EC2 instances before deploying new ones (e.g. using the instance_tags and count_tag parameters), but having to manually figure out this sort of logic for every single resource you manage with Ansible, based on each resource's past history, can be surprisingly complicated (e.g. finding existing instances not only by tag, but also image version, availability zone, etc). This highlights two major problems with procedural IAC tools:

> When dealing with procedural code, the state of the infrastructure is *not* fully captured in the code. Reading through the three Ansible templates we created above is not enough to know what's deployed. You'd also have to know the *order* in which we applied those templates. Had we applied them in a different order, we might end up with different infrastructure, and that's not

> something you can see in the code base itself. In other words, to reason about an Ansible or Chef codebase, you have to know the full history of every change that has ever happened.
>
> The reusability of procedural code is inherently limited because you have to manually to take into account the current state of the codebase. Since that state is constantly changing, code you used a week ago may no longer be usable because it was designed to modify a state of your infrastructure that no longer exists. As a result, procedural code bases tend to grow large and complicated over time.

On the other hand, with the kind of declarative approach used in Terraform, the code always represents the latest state of your infrastructure. At a glance, you can tell what's currently deployed and how it's configured, without having to worry about history or timing. This also makes it easy to create reusable code, as you don't have to manually account for the current state of the world. Instead, you just focus on describing your desired state, and Terraform figures out how to get from one state to the other automatically. As a result, Terraform codebases tend to stay small and easy to understand.

Of course, there are downsides to declarative languages too. Without access to a full programming language, your expressive power is limited. For example, some types of infrastructure changes, such as a rolling, zero-downtime deployment, are hard to express in purely declarative terms. Similarly, without the ability to do "logic" (e.g. if-statements, loops), creating generic, reusable code can be tricky (especially in CloudFormation). Fortunately, Terraform provides a number of powerful primitives—such as input variables, output variables, modules, create_before_destroy, count, and interpolation functions—that make it possible to create clean, configurable, modular code even in a declarative language. We'll discuss these tools more in Part 4, How to create reusable infrastructure with Terraform modules and Part 5, Terraform tips & tricks: loops, if-statements, and pitfalls.

Client/Server Architecture vs Client-Only Architecture

Chef, Puppet, and SaltStack all use a client/server architecture by default. The client, which could be a web UI or a CLI tool, is what you use to issue commands (e.g "deploy X"). Those commands go to a server, which is responsible for executing your commands and storing the state of the system. To execute those commands, the server talks to agents, which must be running on every server you want to configure. This has a number of downsides:

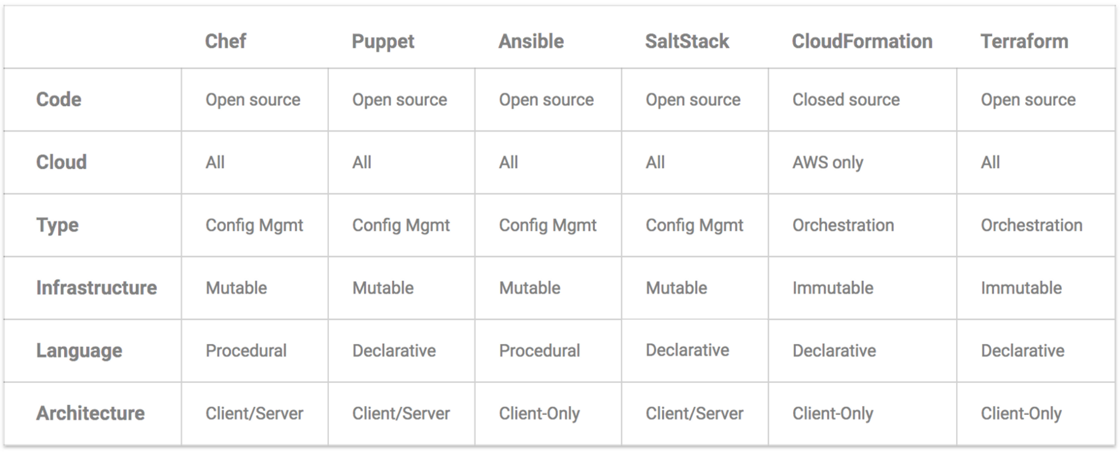
- You have to install and run extra software on every one of your servers.
- You have to deploy an extra server (or even a cluster of servers for high availability) just for configuration management.
- You not only have to install this extra software and hardware, but you also have to maintain it, upgrade it, make backups of it, monitor it, and restore it in case of outages.
- Since the client, server, and agents all need to communicate over the network, you have to open extra ports for them, and configure ways for them to authenticate to each other, all of which increases your surface area to attackers.
- All of these extra moving parts introduce a large number of new failure modes into your infrastructure. When you get a bug report at 3AM, you'll have to figure out if it's a bug in your application code, or your IAC code, or the configuration management client software, or the configuration management agent software, or the configuration management server software, or the ports all those configuration management pieces use to communicate, or the way they authenticate to each other, or…

CloudFormation, Ansible, and Terraform, use a client-only architecture. Actually, CloudFormation is also client/server, but AWS handles all the server details so transparently, that as an end user, you only have to think about the client code. The Ansible client works by connecting directly to your servers over SSH. Terraform uses cloud provider APIs to provision infrastructure, so there are no new authentication mechanisms beyond what you're using with the cloud provider already, and there is no need for direct access to your servers. We found this as the best option in terms of ease-of-use, security, and maintainability.

Conclusion

Putting it all together, below is a table that shows how the most popular

IAC tools stack up:



	Chef	Puppet	Ansible	SaltStack	CloudFormation	Terraform
Code	Open source	Open source	Open source	Open source	Closed source	Open source
Cloud	All	All	All	All	AWS only	All
Type	Config Mgmt	Config Mgmt	Config Mgmt	Config Mgmt	Orchestration	Orchestration
Infrastructure	Mutable	Mutable	Mutable	Mutable	Immutable	Immutable
Language	Procedural	Declarative	Procedural	Declarative	Declarative	Declarative
Architecture	Client/Server	Client/Server	Client-Only	Client/Server	Client-Only	Client-Only

A comparison of popular infrastructure-as-code tools. Click on the image for a larger version. Note that this table shows the "idiomatic" way to use each tool.

At Gruntwork, what we wanted was an open source, cloud-agnostic orchestration tool that supported immutable infrastructure, a declarative language, and a client-only architecture. From the table above, Terraform is the only tool that meets all of our criteria.

Of course, Terraform isn't perfect. It's younger and less mature than all the other tools on the list: whereas Puppet came out in 2005, Chef in 2009, SaltStack and CloudFormation in 2011, and Ansible in 2012, Terraform came out just 2 years ago, in 2014. Terraform is still pre 1.0.0 (latest version is 0.7.4), so there is no guarantee of a stable or backwards compatible API. Bugs are relatively common (e.g. there are over 800 open issues with the label "bug"), although the vast majority are harmless eventual consistency issues that go away when you rerun Terraform. There are also some issues with how Terraform stores state, although there are effective solutions for those issues that we will discuss in Part 3: How to manage Terraform state.

Despite its drawbacks, we find that Terraform's strengths far outshine its weaknesses, and that no other IAC tool fits our criteria nearly as well. If Terraform sounds like something that may fit your criteria too, head over to Part 2: An Introduction to Terraform, to learn more.

For an expanded version of this blog post series, pick up a copy of the book Terraform: Up & Running. If you need help with Terraform, DevOps *practices, or AWS at your company, feel free to reach out to us at Gruntwork. And if you'd like to stay up-to-date with the latest news on DevOps, AWS, software infrastructure, and Gruntwork itself, subscribe to the Gruntwork Newsletter.*