Published in Gruntwork

Yevgeniy Brikman Follow

Sep 26, 2016 · 18 min read · Listen

Save

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

***Update, November 17, 2016**: We took this blog post series, expanded it, and turned it into a book called Terraform: Up & Running!*

***Update, July 8, 2019**: We've updated this blog post series for Terraform 0.12 and released the 2nd edition of Terraform: Up & Running!*

***Update, April 7, 2022:** The early release of the 3rd edition of Terraform: Up & Running, updated through Terraform 1.1, is now available!*

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,

- Mutable Infrastructure vs Immutable Infrastructure
- Procedural vs Declarative
- Master vs Masterless
- Agent vs Agentless
- Large Community vs Small Community
- Mature vs Cutting Edge
- Using Multiple Tools Together

Configuration Management vs Provisioning

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 *provisioning tools*, which means they are designed to provision the servers themselves (as well as the rest of your infrastructure, like load balancers, databases, networking configuration, etc), 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 provisioning tools can do some degree of configuration management. But the focus on configuration management or provisioning 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. 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 a provisioning 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).

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 a provisioning 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 provisioning 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:

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
```

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:

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:

1. 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.
2. The reusability of procedural code is inherently limited because you have to manually 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

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`, and `count`, 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.

Master Versus Masterless

By default, Chef, Puppet, and SaltStack all require that you run a *master server* for storing the state of your infrastructure and distributing updates. Every time you want to update something in your infrastructure, you use a client (e.g., a command-line tool) to issue new commands to the master server, and the master server either pushes the updates out to all the other servers, or those servers pull the latest updates down from the master server on a regular basis.

A master server offers a few advantages. First, it's a single, central place where you can see and manage the status of your infrastructure. Many configuration management tools even provide a web interface (e.g., the Chef Console, Puppet Enterprise Console) for the master server to make it easier to see what's going on. Second, some master servers can run continuously in the background, and enforce your configuration. That way, if someone makes a manual change on a server, the master server can revert that change to prevent configuration drift.

However, having to run a master server has some serious drawbacks:

- **Extra infrastructure**: You have to deploy an extra server, or even a cluster of extra servers (for high availability and scalability), just to run the master.
- **Maintenance**: You have to maintain, upgrade, back up, monitor, and scale the master server(s).
- **Security**: You have to provide a way for the client to communicate to the master server(s) and a way for the master server(s) to communicate with all the other

Chef, Puppet, and SaltStack do have varying levels of support for masterless modes where you just run their agent software on each of your servers, typically on a periodic schedule (e.g., a cron job that runs every 5 minutes), and use that to pull down the latest updates from version control (rather than from a master server). This significantly reduces the number of moving parts, but, as discussed in the next section, this still leaves a number of unanswered
questions, especially about how to provision the servers and install the agent software on them in the first place.

Ansible, CloudFormation, Heat, and Terraform are all masterless by default. Or, to be more accurate, some of them may rely on a master server, but it's already part of the infrastructure you're using and not an extra piece you have to manage. For example, Terraform communicates with cloud providers using the cloud provider's APIs, so in some sense, the API servers are master servers, except they don't require any extra infrastructure or any extra authentication mechanisms (i.e., just use your API keys). Ansible works by connecting directly to each server over SSH, so again, you don't have to run any extra infrastructure or manage extra authentication mechanisms (i.e., just use your SSH keys).

Agent Versus Agentless

Chef, Puppet, and SaltStack all require you to install *agent software* (e.g., Chef Client, Puppet Agent, Salt Minion) on each server you want to configure. The agent typically runs in the background on each server and is responsible for
installing the latest configuration management updates.

This has a few drawbacks:

- **Bootstrapping**: How do you provision your servers and install the agent software on them in the first place? Some configuration management tools kick the can down the road, assuming some external process will take care of this for them (e.g., you first use Terraform to deploy a bunch of servers with a VM image that has the agent already installed); other configuration management tools have a special bootstrapping




- **Maintenance**: You have to carefully update the agent software on a periodic basis, being careful to keep it in sync with the master server if there is one. You also have to monitor the agent software and restart it if it crashes.

- **Security**: If the agent software pulls down configuration from a master server (or some other server if you're not using a master), then you have to open outbound ports on every server. If the master server pushes configuration to the agent, then you have to open inbound ports on every server. In either case, you have to figure out how to authenticate the agent to the server it's talking to. All of this increases your surface area to attackers.

Once again, Chef, Puppet, and SaltStack do have varying levels of support for agentless modes (e.g., salt-ssh), but these often feel like they were tacked on as an afterthought and don't always support the full feature set of the configuration management tool. That's why in the wild, the default or idiomatic configuration for Chef, Puppet, and SaltStack almost always includes an agent, and usually a master too.

All of these extra moving parts introduce a large number of new failure modes into your infrastructure. Each time you get a bug report at 3 a.m., you'll have to figure out if it's a bug in your application code, or your IAC code, or the configuration management client, or the master server(s), or the way the client talks to the master server(s), or the way other servers talk to the master server(s), or…

Ansible, CloudFormation, Heat, and Terraform do not require you to install any extra agents. Or, to be more accurate, some of them require agents, but these are typically already installed as part of the infrastructure you're using. For example, AWS, Azure, Google Cloud, and all other cloud providers take care of installing, managing, and authenticating agent software on each of their physical servers. As a user of Terraform, you don't have to worry about any of that: you just issue commands and the cloud provider's agents execute
them for you on all of your servers. With Ansible, your servers need to run
the SSH Daemon, which is common to run on most servers anyway.




inherent quality of the technology itself. The community determines how many people contribute to the project, how many plug-ins,
integrations, and extensions are available, how easy it is to find help online (e.g., blog posts, questions on StackOverflow), and how easy it is to hire someone to help you (e.g., an employee, consultant, or support company).

It's hard to do an accurate comparison between communities, but you can spot some trends by searching online. The table below shows a comparison of popular IAC tools, with data I gathered during May 2019, including whether the IAC tool is open source or closed source, what cloud providers it supports, the total number of contributors and stars on GitHub, how many commits and active issues there were over a one-month period from mid April to mid May, how many open source libraries are available for the tool, the number of questions listed for that tool on StackOverflow, and the number of jobs that mention the tool on Indeed.com.

	Source	Cloud	Contributors	Stars	Commits (1 month)	Bugs (1 month)	Libraries	StackOverflow	Jobs
Chef	Open	All	562	5,794	435	86	3,832[a]	5,982	4,378[b]
Puppet	Open	All	515	5,299	94	314[c]	6,110[d]	3,585	4,200[e]
Ansible	Open	All	4,386	37,161	506	523	20,677[f]	11,746	8,787
SaltStack	Open	All	2,237	9,901	608	441	318[g]	1,062	1,622
CloudFormation	Closed	AWS	?	?	?	?	377[h]	3,315	2,318
Heat	Open	All	361	349	12	600[i]	0[j]	88	2,201[k]
Terraform	Open	All	1,261	16,837	173	204	1,462[l]	2,730	3,641

[a] This is the number of cookbooks in the Chef Supermarket.
[b] To avoid false positives for the term "chef", I searched for "chef devops".
[c] Based on the Puppet Labs JIRA account.
[d] This is the number of modules in Puppet Forge.
[e] To avoid false positives for the term "puppet", I searched for "puppet devops".
[f] This is the number of reusable roles in Ansible Galaxy.
[g] This is the number of formulas in the Salt Stack Formulas GitHub account.
[h] This is the number of templates in the awslabs GitHub account.
[i] Based on the OpenStack bug tracker.
[j] I could not find any collections of community Heat templates.
[k] To avoid false positives for the term "heat", I searched for "openstack".

Obviously, this is not a perfect apples-to-apples comparison. For example, some of the tools have more than one repository, and some use other methods for bug tracking and questions; searching for jobs with common words like "chef" or "puppet" is tricky; Terraform split the provider code out into separate repos in 2017, so measuring activity on solely the core repo dramatically understates activity (by at least 10x); and so on.

That said, a few trends are obvious. First, all of the IAC tools in this comparison are open source and work with many cloud providers, except for CloudFormation, which is closed source, and only works with AWS. Second, Ansible leads the pack in terms of popularity, with Salt and Terraform not too far behind.

Another interesting trend to note is how these numbers have changed since the 1st version of this blog post series. The table below shows the percent change in each of the numbers from the values I gathered back in September, 2016.

	Source	Cloud	Contributors	Stars	Commits (1 month)	Issues (1 month)	Libraries	StackOverflow	Jobs
Chef	Open	All	+18%	+31%	+139%	+48%	+26%	+43%	-22%
Puppet	Open	All	+19%	+27%	+19%	+42%	+38%	+36%	-19%
Ansible	Open	All	+195%	+97%	+49%	+66%	+157%	+223%	+125%
SaltStack	Open	All	+40%	+44%	+79%	+27%	+33%	+73%	+257%
CloudFormation	Closed	AWS	?	?	?	?	+57%	+441%	+249%
Heat	Open	All	+28%	+23%	-85%	+1,566%	0	+69%	+2,957%
Terraform	Open	All	+93%	+194%	-61%	-58%	+3,555%	+1,984%	+8,288%

How the IAC communities have changed between September, 2016 and May, 2019. Click for the full-size image.

Again, the data here is not perfect, but it's good enough to spot a clear trend: Terraform and Ansible are experiencing explosive growth. The increase in the number of contributors, stars, open source libraries, StackOverflow posts, and jobs is through the roof (note: the decline in Terraform's commits and issues is solely due to the fact that I'm only measuring the core Terraform repo, whereas in 2017, all the

Mature Versus Cutting Edge

Another key factor to consider when picking any technology is maturity. The table below shows the initial release dates and current version number (as of May, 2019) for of each of the IAC tools.

	Initial release	Current version
Puppet	2005	6.0.9
Chef	2009	12.19.31
CloudFormation	2011	2010-09-09
SaltStack	2011	2019.2.0
Ansible	2012	2.5.5
Heat	2012	12.0.0
Terraform	2014	0.12.0

A comparison of IAC maturity as of May, 2019. Click for the full-size image.

Again, this is not an apples-to-apples comparison, since different tools have different versioning schemes, but some trends are clear. Terraform is, by far, the youngest IAC tool in this comparison. It's still pre 1.0.0, so there is no guarantee of a stable or backward compatible API, and bugs are relatively common (although most of them are minor). This is Terraform's biggest weakness: although it has gotten extremely popular in a short time, the price you pay for using this new, cutting-edge tool is that it is not as mature as some of

Although I've been comparing IAC tools this entire blog post, the reality is that you will likely need to use multiple tools to build your infrastructure. Each of the tools you've seen has strengths and weaknesses, so it's your job to pick the right tool for the right job.

Here are three common combinations I've seen work well at a number of companies:

1. Provisioning plus configuration management
2. Provisioning plus server templating
3. Provisioning plus server templating plus orchestration

Provisioning plus configuration management

Example: Terraform and Ansible. You use Terraform to deploy all the underlying infrastructure, including the network topology (i.e., VPCs, subnets, route tables), data stores (e.g., MySQL, Redis), load balancers, and servers. You then use Ansible to deploy your apps on top of those servers.

This is an easy approach to start with, as there is no extra infrastructure to run (Terraform and Ansible are both client-only applications) and there are many ways to get Ansible and Terraform to work together (e.g., Terraform adds special tags to your servers and Ansible uses those tags to find the server and configure them). The major downside is that using Ansible typically means you're writing a lot of procedural code, with mutable servers, so as your code base, infrastructure, and team grow, maintenance may become more difficult.

Provisioning plus server templating

Example: Terraform and Packer. You use Packer to package your apps as virtual machine images. You then use Terraform to deploy (a) servers with these virtual machine images and (b) the rest of your infrastructure, including the network topology (i.e., VPCs, subnets, route tables), data stores (e.g., MySQL, Redis), and load balancers.

This is also an easy approach to start with, as there is no extra infrastructure to run

will slow down your iteration speed. Second, the deployment strategies you can implement with Terraform are limited (e.g., you can't implement blue-green deployment natively in Terraform), so you either end up writing lots of complicated deployment scripts, or you turn to orchestration tools, as described next.

Provisioning plus server templating plus orchestration

Example: Terraform, Packer, Docker, and Kubernetes. You use Packer to create a virtual machine image that has Docker and Kubernetes installed. You then use Terraform to deploy (a) a cluster of servers, each of which runs this virtual machine image and (b) the rest of your infrastructure, including the network topology (i.e., VPCs, subnets, route tables), data stores (e.g., MySQL, Redis), and load balancers. Finally, when the cluster of servers boots up, it
forms a Kubernetes cluster that you use to run and manage your Dockerized applications.

The advantage of this approach is that Docker images build fairly quickly, you can run and test them on your local computer, and you can take advantage of all the built-in functionality of Kubernetes, including various deployment strategies, auto healing, auto scaling, and so on. The drawback is the added complexity, both in terms of extra infrastructure to run (Kubernetes clusters are difficult and expensive to deploy and operate, though most major cloud
providers now provide managed Kubernetes services, which can offload some of this work), and in terms of several extra layers of abstraction (Kubernetes, Docker, Packer) to learn, manage, and debug.

Conclusion

Putting it all together, the table below shows how the most popular IAC tools stack up. Note that this table shows the *default* or *most common* way the various IAC tools are used, though as discussed earlier, these IAC tools are flexible enough to be used in other configurations, too (e.g., Chef can be used without a master, Salt can be used to do immutable infrastructure).

	Source	Cloud	Type	Infrastructure	Language	Agent	Master	Community	Maturity
Chef	Open	All	Config Mgmt	Mutable	Procedural	Yes	Yes	Large	High
Puppet	Open	All	Config Mgmt	Mutable	Declarative	Yes	Yes	Large	High
Ansible	Open	All	Config Mgmt	Mutable	Procedural	No	No	Huge	Medium
SaltStack	Open	All	Config Mgmt	Mutable	Declarative	Yes	Yes	Large	Medium
CloudFormation	Closed	AWS	Provisioning	Immutable	Declarative	No	No	Small	Medium
Heat	Open	All	Provisioning	Immutable	Declarative	No	No	Small	Low
Terraform	Open	All	Provisioning	Immutable	Declarative	No	No	Huge	Low

A comparison of the most common way to use the most popular IAC tools. Click for the full-size image.

At Gruntwork, what we wanted was an open source, cloud-agnostic provisioning 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. It's certainly not perfect, especially in terms of maturity, but 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 (2nd edition available now!). If you need help with Terraform, DevOps practices, or AWS at your company, feel free to reach out to us at Gruntwork.

Thanks to Josh Padnick