Exploring Different Ways of Implementing Infrastructure as Code with SaltStack and Terraform

Robin Bråtfors

Automated Software Testing and DevOps - DD2482

2020-05-31

1 Introduction

Working in a large team and developing software can be chaotic, each member might work on their own thing and if communication is lacking, there might be inconsistencies with the development environments. If someone in the team maintains the configuration of their own environments, these environments are bound to become unique over time as new, local updates are made. Their software might only work on their unique environments. To combat this, any local changes need to be spread to every other environment so that everything is in sync. Without automation you would of course have to fix this manually, something which can take time and also cause additional problems from human error, even more-so if the changes are poorly documented and hard to reproduce.

A solution to this problem is Infrastructure as Code (IaC). This paper will explore how IaC relates to DevOps and what kind of benefits it can bring to a team. It will also examine two kinds how tools, SaltStack and Terraform, that achieve IaC in different ways.

2 What is IaC?

IaC is the practice of specifying and automating system configuration and deployment through the use of code, which allows maintenance of the system configuration through traditional code practices [1].

IaC grants the infrastructure some of the benefits that normal code has. If the configuration needs to be scaled up, i.e. by adding another server, the code will be updated and the change can be traced if version control is employed [2]. This makes it easier for everyone in the team to track any modifications made to the infrastructure, you can see who did what and when they did it.

Another benefit of IaC is that the infrastructure becomes well defined and reproducible. The configuration files are consistent and can be deployed again and again without inconsistencies between systems [3]. This also makes the process of deploying new environments much faster. Instead of trying to manually set up 10 identical machines, you can just use a script that creates them instantly. This grants increased reusability of the infrastructure, with the added benefit of automation.

IaC tools can build the infrastructure in different ways, they are often categorized into two groups: declarative and imperative, though overlapping does occur [3]. Imperative is sometimes called procedural [4]. The difference between the two is that imperative tools gives the exact sequence of instructions that the infrastructure needs to reach a desired state, while declarative tools just outline the desired state and let the provider reach it on its own [3]. Imperative can be seen as giving someone exact directions to reach your house from their starting point, while declarative just gives them the address.

The declarative approach thus gives more flexibility in how the final state can be reached and the infrastructure becomes easier to scale, since you do not need to add how new components should be created. On the other hand, the procedural approach gives you more control in how the infrastructure should be created. Terraform uses the declarative approach while Salt uses both the declarative and the imperative approach.

Another difference between tools is the mutability of the infrastructure. Some tools like Salt see infrastructure components as mutable, which means that every time the configuration changes, those changes

are made to the already existing components. Other tools such as Terraform see the components as immutable, signifying that the already existing components cannot be changed and that the changes have to be implemented by creating new components that replace the already existing ones [5].

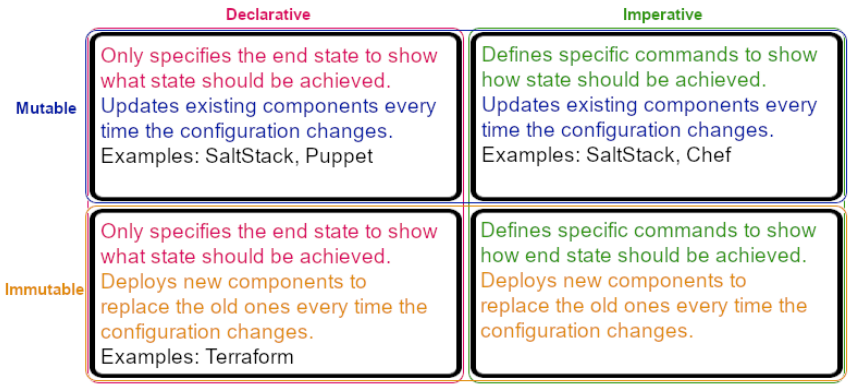

Figure 1: approaches and mutability.

3 Configuration Management vs. Orchestration

The purpose of configuration management tools are to install, update, and manage the software on existing infrastructure components, such as servers [4]. It allows you to implement features such as version control to make it easier to manage and replicate servers [6].

The definition of orchestration is a little muddled. In his book about Terraform and IaC, Brikman separates orchestration and provisioning [4]. He regards orchestration tools to be tools such as Kubernetes or Docker Swarm that mostly deal with containers, while provisioning tools are the only tools that are primarily tasked with actually creating servers.

In contrast, in her IaC miniseries, Woods considers provisioning to be a part of orchestration, a definition where orchestration tools also have the task of organizing and managing the infrastructure components they provision [5].

These conflicting definitions probably stems from the authors omitting what kind of orchestration they are writing about. Brikman is of course talking about *container*-orchestration, while Woods might be thinking of it in a broader sense as something akin to *infrastructure*-orchestration.

It does not help that there exists quite a bit of overlap in what the tools can do, e.g. a primarily configuration management tool like Ansible has some orchestration capabilities, such as being able to deploy servers by itself [7].

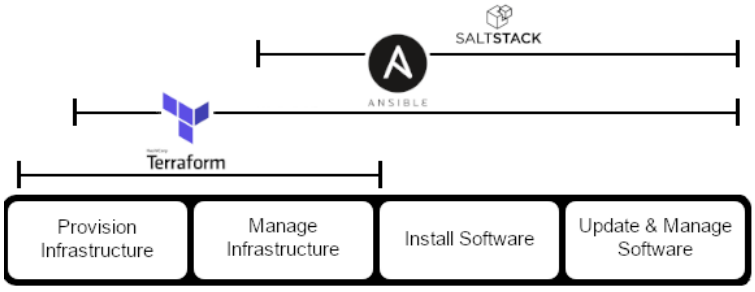

Figure 2: tool functions.

This paper will treat provisioning as part of orchestration and thus Terraform will be considered an orchestration tool.

3.1 SaltStack

SaltStack, also known as Salt, is an open-source, configuration management tool written in Python [8]. It was created in 2011 and has a large community with over 2200 contributors and around 11k stars on their GitHub repository [9].

The main parts of Salt are the Salt master and the Salt minion. The master works as a command center that can give out instructions to the minions, which are all connected to it. This grants the benefits of having a centralized infrastructure, which makes it easier to manage [8].

Though master servers also have some detriments, they require more infrastructure since you need more servers to run the master, and having more servers means more maintenance. Another disadvantage is that the communication between master and minions require opening more ports which gives potential attackers more surface area to work with [4].

SaltStack uses salt terminology to describe some of its features, for instance salt *grains* are pieces of information that relate to the operating system, IP address and other system properties [10]. Salt pillars, on the other hand, contain more static data and can be used to distribute configuration values to minions [11]. Lastly, the salt mine is used to archive arbitrary data that is returned from minions, it is returned to the master and only the most recent values are stored [12].

The way you interact with Salt is through command-line interface tools, the most used tool is `salt`. With it you can execute functions on all systems or on specific targets. Targeting is a flexible way of controlling what systems you want to modify. Using `'*'` as the target will apply the called function on all managed systems [13]. The targets can be multiple or single, they can be filtered, e.g. `'minion[0-5]'`, or they can be explicitly stated, e.g. `'minion1,minion2'`. Targeting is also able to use Boolean operators such as `and`, `or` and `not` for more advanced matching [14]. For instance, the `not` operator can be used to target all minions that are not running a certain operating system [8].

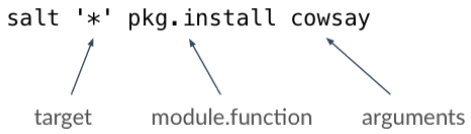

Figure 3: example of Salt command syntax [13].

3.1.1 Remote Execution

With Salt you can execute commands on one or several remote systems concurrently. This is done by using execution modules, which are just Python modules designed for Salt execution [15]. Salt execution modules can do a great deal of tasks and functionalities [16]. Salt has a wide variety of modules available in their online documentation [17]. With them, Salt can install packages, manage services or monitor minion states, among other things.

3.1.2 Configuration Management

Configuration management in Salt expands on the remote execution feature to create replicable configuration for all minions [8]. As with remote execution, we utilize Python modules but for this purpose they are called state modules. The state modules contain functions that describe the final state of the system, together these functions can create state files [15].

The state files can be represented in different ways but the default format is YAML. The files are laid out in a hierarchy in the Salt file server, the hierarchy is called a state tree. The top file in the state tree is essential and controls which of the state files should be applied to which minions [18]. Here is an example of a typical state file:

```
apache:                 # ID declaration
  pkg.installed: []     # Module function
```

```
service.running:      # Module function
  - require:          # Requisite statement
    - pkg: apache     # Service only starts after the package has been installed
```

3.2 Terraform

Brikman describes Terraform as an open source tool developed by HashiCorp, written in the programming language Go. It was created in 2014 and has a healthy community with around 1400 contributors and about 22.2k stars on their repository on GitHub [19].

The written code compiles into a binary that can be used to deploy infrastructure. It works by making API calls to a provider such as AWS or Google Cloud. The API calls are created from text files called *Terraform configurations*, you can specify all of your infrastructure within these files and thus have all of it centralized in one place. The files can be committed to version control like normal code files [4]. This makes it easier to manage the infrastructure in terms of versioning and its re-usability.

3.2.1 States

Before implementing the changes you want, it is a good idea to first look over the execution plan, this is done with the `terraform plan` command. The command creates or updates the plan. It will show a list of all the planned changes needed to reach the final state as defined in the configuration, which allows you to make sure it matches your intentions before you decide to apply them [20]. It could look something like this:

```
Terraform will perform the following actions:

  # azurerm_resource_group.rg will be created
  + resource "azurerm_resource_group" "rg" {
      + id       = (known after apply)
      + location = "westus"
      + name     = "myTFResourceGroup"
      + tags     = (known after apply)
    }
```

To apply the actions in the plan, you simply use the `terraform apply` command [21]. Something to note is that the apply command includes the plan command, but it can be nice to use the plan command separately for quick checks. Since Terraform uses the declarative approach, this is all that's needed to fetch create new infrastructure, no specification from the user is required.

An important part of Terraform configurations are state files. The state of your infrastructure will be stored in a state file and even though the file is in JSON format, direct file editing is not recommended. Instead you should use the `terraform state` command to modify the state using the command-line interface [22].

Every time you want to implement changes to the infrastructure with the apply command, the state file gets updated and any new actions that are needed are determined, then those actions are executed. [23].

The way that Terraform refreshes states and forms plans is by building a dependency graph from the Terraform configurations, this graph is then walked with a depth-first traversal to determine the order that the resources should be built, i.e. how the plan should look [24].

Terraform provides an easy way to see how the dependency graph looks with the command `terraform graph`, the command can also have added arguments such as highlighting cycles [25]. Something which is good to notice since cycles mess up the evaluation order in dependency graphs.

A problem with state files is that they are susceptible to concurrent changes from multiple team members that could result in conflicts, this can be solved by using backends with locking features, e.g. Azure Storage or AWS S3 [26][23]. With locking, the state will be locked for any operation that could change it. Locking is enabled automatically after every successful write operation and should also unlock automatically when the file is free [27].

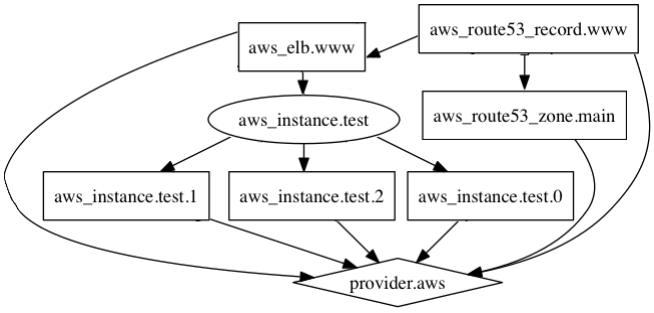

Figure 4: example graph output [25].

3.3 SaltStack vs. Terraform

The core feature of Salt is its remote execution and its master-minion topology. These features allow users to configure a large number of hosts at the same time. Salt provides a lot of functionality in the form of its modules making it easy to install packages and manage your hosts.

Having a master-minion topology grants some benefits such as centralizing your infrastructure and granting a place to manage it all. However, you'll have to deploy extra servers to support the master which will add maintenance and more security risks that you need to consider. You also need to monitor the minions and make sure that they are in sync with the master.

Terraform grants you highly reusable and shareable infrastructure thanks to its declarative approach and simple state files, these files can easily be shared between teams or projects. Its execution plans provide you with concrete information that is really helpful while planning your infrastructure, it ensures that you know exactly what Terraform will do when provisioning components. The feature to build and visualize the dependency graph of your infrastructure makes it easier for you to get an overlook of the infrastructure and how different components relate to each other.

Since Terraform is a relatively new tool, it is still only on its initial release version of 0.x which means it maturity is quite low. Encountering the occasional bug is to be expected with any new software and Terraform is no exception.

Both tools have large and active communities which can provide support for issues that your team encounters, but Terraform seems to have a faster growing user base and community.

When it comes to choosing between these two tools, the focus should be less on the minutiae features of the tools and more on what areas they specialize in and what your team needs. If your team has all the servers it could ever need already set up, but you still need to constantly maintain and install new packages on them then SaltStack is the clear choice.

If your team instead has problems with scaling your infrastructure up and down depending on spikes in user activity then Terraform will be of tremendous help thanks to its great provisioning capabilities.

4 Conclusion

The two tools covered in this paper are quite capable and it would be perfectly fine to just use one of them, but they do excel in specific areas which might incline you to use both of them at the same time. Terraform was created for orchestration and that is what it does best. In the same vein, Salt really shines when it comes to configuration management. If you do use both of them then your infrastructure will be reusable, flexible and easy to maintain.

IaC brings a lot of coding benefits to your infrastructure. Being able to implement version control and have that history over changes is a huge asset for any team. To see who did what and when they did it just reduces a lot of headaches when it comes to finding bugs and other conflicts.

To be able to have a consistent configuration is a great boon, when you want to reproduce the infrastructure, you will get a carbon copy that executes and behaves the same way every time. With this, it becomes easier to scale your infrastructure to fit your current needs. You can scale up the infrastructure during peak hours and then scale it down during the night to save money on maintenance.

Lastly, nobody likes repetitive and dull tasks. Humans should not have to focus on these kinds of tasks, they require no creativity and are often error-prone. IaC removes those tasks with automation, giving people the freedom to work on more exciting things that require a human touch.

All these benefits really speeds up the work of the whole team. You will find bugs faster with version control. You can reuse infrastructure easier and can have exact copies up and running in no time. You will waste less time working on things you should not have to do. In conclusion, treating your code as infrastructure saves a lot of time in both the developing part and the operations part of DevOps, it's a worthwhile endeavor for any team.

References

[1] Jez Humble & David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[2] Christopher Null. *Infrastructure as code: The engine at the heart of DevOps*. 2017. URL: https://techbeacon.com/enterprise-it/infrastructure-code-engine-heart-devops.

[3] Carlos Schults. *What Is Infrastructure as Code? How It Works, Best Practices, Tutorials*. 2019. URL: https://stackify.com/what-is-infrastructure-as-code-how-it-works-best-practices-tutorials/.

[4] Yevgeniy Brikman. *Terraform: Up & Running: Writing Infrastructure as Code*. O'Reilly Media, 2019.

[5] Emily Woods. *Infrastructure as Code, Part One*. 2018. URL: https://crate.io/a/infrastructure-as-code-part-one/.

[6] Erika Heidi. *An Introduction to Configuration Management*. 2019. URL: https://www.digitalocean.com/community/tutorials/an-introduction-to-configuration-management.

[7] SpinupWP. *Automating Server Setup with Ansible*. 2016. URL: https://spinupwp.com/automating-server-setup-ansible/.

[8] Colton Myers. *Learning SaltStack*. Packt Publishing Ltd, 2016.

[9] SaltStack. *What is SaltStack?* 2020. URL: https://github.com/saltstack/salt.

[10] SaltStack. *Grains*. 2020. URL: https://docs.saltstack.com/en/latest/topics/grains/.

[11] SaltStack. *Storing Static Data in the Pillar*. 2020. URL: https://docs.saltstack.com/en/latest/topics/pillar/.

[12] SaltStack. *The Salt Mine*. 2020. URL: https://docs.saltstack.com/en/latest/topics/mine/.

[13] SaltStack. *Execute Commands*. 2020. URL: https://docs.saltstack.com/en/getstarted/fundamentals/remotex.html.

[14] SaltStack. *Targeting*. 2020. URL: https://docs.saltstack.com/en/getstarted/fundamentals/targeting.html.

[15] Emily Woods. *Infrastructure as Code, Part Three: Configuration With Salt*. 2019. URL: https://crate.io/a/infrastructure-as-code-configuration-with-salt/.

[16] SaltStack. *Remote execution*. 2020. URL: https://docs.saltstack.com/en/latest/topics/execution/index.html.

[17] SaltStack. *Execution Modules*. 2020. URL: https://docs.saltstack.com/en/latest/ref/modules/all/index.html#all-salt-modules.

[18] SaltStack. *State System Reference*. 2020. URL: https://docs.saltstack.com/en/latest/ref/states/index.html#state-system-reference.

[19] HashiCorp. *Terraform*. 2020. URL: https://github.com/hashicorp/terraform.

[20] HashiCorp. *Command: plan*. 2020. URL: https://www.terraform.io/docs/commands/plan.html.

[21] HashiCorp. *Command: apply*. 2020. URL: https://www.terraform.io/docs/commands/apply.html.

[22] HashiCorp. *State*. 2020. URL: https://www.terraform.io/docs/state/index.html.

[23] Emily Woods. *Infrastructure as Code, Part Two: A Closer Look at Terraform*. 2019. URL: https://crate.io/a/infrastructure-as-code-part-two-a-closer-look-at-terraform/.

[24] HashiCorp. *Resource Graph*. 2020. URL: https://www.terraform.io/docs/internals/graph.html.

[25] HashiCorp. *Command: graph*. 2020. URL: https://www.terraform.io/docs/commands/graph.html.

[26] HashiCorp. *Backends*. 2020. URL: https://www.terraform.io/docs/backends.

[27] HashiCorp. *State locking*. 2020. URL: https://www.terraform.io/docs/state/locking.html.