

A comparison of the benefits of Terraform and CloudFormation in different scenarios

Author:
Vasil Petrov
Cloud Infrastructure Engineer
HeleCloud

March, 2020

Table of Contents

Abstract	3
Introduction	3
1. Language	3
2. Multi-Cloud Compatibility	4
3. Visibility	5
4. Flexibility	6
5. Drift Detection	6
6. Adopting Existing Cloud Resources	7
7. Terraform Modules vs. Nested Stacks	7
8. Multiple Accounts / Region Deployments	8
9. Zero-time Deployments	8
10. Available Resources	9
11. Nice-to-Have/ Advanced Features	9
Conclusion	10
References	10

Abstract

The purpose of this whitepaper is to provide insight into the capabilities and strengths of HashiCorp Terraform and AWS CloudFormation in different scenarios and use cases.

Introduction

As a Cloud consulting company, choosing the right tools for building and maintaining Infrastructure as Code (IaC) environments is critical for our business. Our main area of experience is the Amazon Web Services (AWS) platform, the dominant Public Cloud Services Provider.

We will consider in the table below a number of areas below for the purpose of comparing and conclude with the selection of the best tool for Infrastructure and Code – AWS CloudFormation or HashiCorp Terraform.

1. Language

HashiCorp Terraform	AWS CloudFormation
<ul style="list-style-type: none"> ■ Terraform uses HashiCorp Configuration Language (HCL) developed to strike a balance between human-readable and machine-friendly languages. ■ Terraform has a rich set of string interpolations and built-in functions, including conditionals and loops, which allow modelling complex logic in Domain-Specific Language (DSL) without having to resort to a fully-fledged programming language. 	<ul style="list-style-type: none"> ■ From a programming point of view, CloudFormation allows using common programming languages, such as Python, Java, Ruby Go and more via the AWS software development kits to interact with the CloudFormation API. ■ The core pieces of CloudFormation are simple text file written in JSON or YAML.

- Code snippet of the HCL:

```
resource "aws_instance" "instance" {
  count          = 2
  availability_zone = data.aws_availability_zones.available.names[0]
  ami            = data.aws_ami.coreos.image_id
  instance_type  = "t2.micro"
  security_groups = [aws_security_group.web.name]
  key_name       = aws_key_pair.kp.key_name
  user_data      = data.template_file.web_cloud_config.rendered
}
```

- Code snippet of CloudFormation JSON configuration template:

```
{
  "Resources": {
    "Ec2Instance": {
      "Type": "AWS::EC2::Instance",
      "Properties": {
        "ImageId": {
          "Fn::FindInMap": [
            "AWSRegionArch2AMI",
            {
              "Ref": "AWS::Region"
            }
          ],
          "Fn::FindInMap": [
            "AWSInstanceType2Arch",
            {
              "Ref": "InstanceType"
            }
          ],
          "Arch"
        }
      ]
    },
    "KeyName": {
      "Ref": "KeyName"
    },
    "InstanceType": {
      "Ref": "InstanceType"
    },
    "SecurityGroups": [
      {
        "Ref": "Ec2SecurityGroup"
      }
    ],
    "BlockDeviceMappings": [
      {
        "DeviceName": "/dev/sda1",
        "Ebs": {
          "VolumeSize": "50"
        }
      },
      {
        "DeviceName": "/dev/sdm",
        "Ebs": {
          "VolumeSize": "100"
        }
      }
    ]
  }
}
```

2. Multi-Cloud Compatibility

HashiCorp Terraform	AWS CloudFormation
<ul style="list-style-type: none"> ■ Terraform can support multiple providers including AWS, Microsoft Azure, Google Cloud, OpenStack and Alibaba Cloud. ■ Having a single tool managing resources spread across multiple Cloud environments limits the overhead of embedding different IaC tools for different Cloud providers. ■ For those that need a general-purpose tool, Terraform is likely to cut the muster better. 	<ul style="list-style-type: none"> ■ AWS CloudFormation is a native IaC service for provisioning and managing infrastructure resources. ■ As a tool it is more suitable to run only with AWS; managing existing CloudFormation templates or using AWS Managed Services.

3. Visibility

HashiCorp Terraform

- With versions release 0.12.x, Terraform had made a huge improvement to the error messages.
- It provides you with the exact line of code where the error resides and a snippet of the terraform code.
- With the improved error messages, detecting, debugging and resolving issues is much easier now.

AWS CloudFormation

- As a managed AWS service, CloudFormation has limited visibility.
- It allows for the rolling back changes when is unable to deploy successfully all resources defined in the CloudFormation stack template.

- Example output of Terraform error messages:

```
Error: Error in function call

on main.tf line 16, in data "template_file" "web_cloud_config":
16:   template = file("${path.module}/web_cloud_config.sh")
   |-----
   | path.module is "."

Call to function "file" failed: no file exists at web_cloud_config.sh.

Error: Unsupported block type

on main.tf line 18, in data "template_file" "web_cloud_config":
18:   vars {

Blocks of type "vars" are not expected here. Did you mean to define argument "vars"? If so, use the equals sign to assign it a value.
```

- Example of CloudFormation rollback feature:

Events New events available			
<input type="text" value="Search events"/>			
Timestamp	Logical ID	Status	Status reason
2019-11-22 14:56:28 UTC+0200	rollingupdate	⊗ UPDATE_ROLLBACK_IN_PROGRESS	The following resource(s) failed to update: [ELBSecurityGroup].
2019-11-22 14:56:27 UTC+0200	ELBSecurityGroup	⊗ UPDATE_FAILED	Invalid value '-11' for IP protocol. Unknown protocol number. (Service: AmazonEC2; Status Code: 400; Error Code: InvalidParameterValue; Request ID: 60518792-bb92-42fa-be26-8c24d6410ee2)

4. Flexibility

HashiCorp Terraform	AWS CloudFormation
<ul style="list-style-type: none"> Adapting to changes is one of the strengths of Terraform. Let's take the following use case, defining a "simple" service inside a single file; months later we realise that the "simple" service is not so simple at all. The solution would be to refactor the code into smaller parts, for example in modules. Avoids the need to redeploy the entire infrastructure, because of refactoring your code. 	<ul style="list-style-type: none"> Allows the refactoring of code without factoring any downtime. AWS documentations have rich examples of how to refactor your code step-by-step. In general, people find refactoring JSON or YAML syntax a bit more complicated rather than Terraform HCL We must note that different companies have different preferences for defining resources.

5. Drift Detection


HashiCorp Terraform	AWS CloudFormation
<ul style="list-style-type: none"> Most helpful when teams want to perform drift detection before any changes to be done to the infrastructure. Terraform provides advanced lifecycle hooks to make drift detection possible. 	<ul style="list-style-type: none"> Most appropriate where checking for drifts on every create, update or delete API call is not necessary.

- Example of CloudFormation drift detection output:

Stack drift status


Drift detection enables you to detect whether a stack's actual configuration differs, or has drifted, from its template configuration. [Learn more](#)

Drift status

 DRIFTED











Last drift check time

2019-11-22 17:07:56 UTC+0200

 Only resources which currently support drift detection are displayed here. To view all of your stack resources, see your stack details page. [Learn more](#)

Resource drift status (5)

[View drift details](#)[Detect drift for resource](#)

Logical ID	Physical ID	Type	Drift status	Timestamp
 AutoScalingGroup	rollingupdate-AutoScalingGroup-34BD5HYSTBMO	AWS::AutoScaling::AutoScalingGroup	 IN_SYNC	2019-11-22 17:07:55 UTC+0200
 EC2SecurityGroup	sg-0db7d658235e22738	AWS::EC2::SecurityGroup	 MODIFIED	2019-11-22 17:07:55 UTC+0200
 ELB	elb	AWS::ElasticLoadBalancing::LoadBalancer	 IN_SYNC	2019-11-22 17:07:55 UTC+0200
 ELBSecurityGroup	sg-0779e9befc9dd5535	AWS::EC2::SecurityGroup	 IN_SYNC	2019-11-22 17:07:56 UTC+0200
 LaunchConfiguration	rollingupdate-LaunchConfiguration-1PLNQ98I4HOW	AWS::AutoScaling::LaunchConfiguration	 IN_SYNC	2019-11-22 17:07:55 UTC+0200

6. Adopting Existing Cloud Resources

HashiCorp Terraform	AWS CloudFormation
<ul style="list-style-type: none">■ Terraform is suitable in cases where we want to adopt manually created Cloud resource under Terraform management.■ Terraform gives the team the ability to adopt existing Cloud resources, which were created manually and import them into the state file. We can start managing existing Cloud resources without the need for any downtime. The tricky part here is that we must prepare the necessary code, because if we don't do it, the next time we execute plan or apply commands, terraform will want to destroy them.	<ul style="list-style-type: none">■ Until recently CloudFormation was not able to adopt existing resources but managed to catch up with this great feature.■ Most of the steps during the importing of an existing Cloud resource are done through the CloudFormation GUI.

7. Terraform Modules vs. Nested Stacks

HashiCorp Terraform	AWS CloudFormation
<ul style="list-style-type: none">■ Modules are code template are easily shared between multiple stacks. With Terraform, we can place our modules in a git repository and use semantic versioning to reference it. Anyone with access to that git repository can reuse the common terraform modules.	<ul style="list-style-type: none">■ Nested stack files are stored to put them inside S3 bucket, which lacks all the familiar benefits.

8. Multiple Accounts / Region Deployments

HashiCorp Terraform	AWS CloudFormation
<p>Terraform provides few options that allows us to achieve multiple account and/or regions deployments:</p> <ul style="list-style-type: none"> ■ You can have different states and variables configurations for each account(environment). That are provided in during each terraform apply execution. ■ Terraform provider alias is a functionality that allows you to define all your accounts and related roles that will be used for deployments and then run pre-defined modules against thus aliases. ■ Terraform workspaces is a concept previously known as "environments" in version 0.9. This feature provides the ability single configuration to have multiple named workspaces, allowing multiple states to be associated with a that single terraform configuration. 	<ul style="list-style-type: none"> ■ In July 2017, AWS announced CloudFormation StackSets in response of the high demand customer requests for ability to deploy CloudFormation Stacks from one central place to multiple accounts and/or regions. ■ StackSets enable you to create, update, or delete stacks across multiple accounts and regions in a single operation. <p>Using an administrator account, you define and manage an AWS CloudFormation template, and use the template as the basis for provisioning stacks into selected target accounts across specified regions.</p>

9. Zero-time Deployments

HashiCorp Terraform	AWS CloudFormation
<ul style="list-style-type: none"> ■ Although Terraform can perform zero-time deployments thangs to null resources and provisioners (local-exec and remote-exec), the official documentation recommends using them as a last resort. ■ The reason why Terraform is not recommending using null resources and provisioners is that Terraform loses context what the provisioner would do to the local/ remote machine and what would be the outcome. 	<ul style="list-style-type: none"> ■ CloudFormation is suitable where zero-time deployments are required. A great example would be rolling updates on AWS with Elastic Load Balancer. ■ CloudFormation zero-time deployment is possible thangs to python helper scripts, in particular "cfn-signal" helper script, which waits for an event to happen for the deployment to continue.

10. Available Resources

HashiCorp Terraform	AWS CloudFormation
<ul style="list-style-type: none"> ■ Terraform is suitable in cases where we want as many available resources as possible out-of-the-box, without any additional effort from our side. From the official documentation we could see a rich set of available AWS resources. ■ Terraform is able to update its providers as soon as there is available or updated SDK. Also, we could always check if there is a new version of the provider by executing "terraform init" command. 	<ul style="list-style-type: none"> ■ CloudFormation, on the other hand, has less available resources out-of-the-box, but has a secret weapon called – "CloudFormation Custom Resource". ■ A custom resource is a Lambda function, which interacts with an AWS API and allows us to manage a resource before CloudFormation to officially add support for it. ■ The lambda function can be written in Java, Go, PowerShell, Node.js, C#, Python and Ruby

11. Nice-to-Have/ Advanced Features

HashiCorp Terraform	AWS CloudFormation
<ul style="list-style-type: none"> ■ Includes Policy as Code (PaC) methodology for our Infrastructure as Code environment. ■ PaC can ensure compliance standards such as PCI-DSS, SOC or GDPR against our IaC environment before any resources to be deployed. ■ Also, PaC can enforce security policies such as ensuring only certain applications to run on a public network or expose specific ports to the Internet. ■ Currently, Sentinel the HashiCorp tool for Policy as Code is available and embedded only in Terraform enterprise products, an example would be Terraform Cloud and Terraform Enterprise. 	<ul style="list-style-type: none"> ■ Use CFN Linter, an open source tool that helps us to validate your templates not only against syntax, but also checking valid values for resource properties, best practices or custom checks. ■ CloudFormation is suitable for people which have a solid programming background. CloudFormation has an AWS Cloud Development Kit (CDK), which allows deploying Cloud infrastructure using the programming language, which we feel most comfortable. ■ AWS CDK is a software development framework for defining Cloud infrastructure in code and provisioning it through AWS CloudFormation in the background. ■ Supported programming languages for AWS CDK are: <ul style="list-style-type: none"> ■ JavaScript ■ TypeScript ■ Python ■ Java ■ C# ■ Ability to deploy applications using the Serverless Framework.

Conclusion

When deciding between two powerful Infrastructure as Code tools, choosing one over the other is a difficult decision. What it comes down to is the confidence and comfort for the user of using each tool. Personally, as a business we prefer to use Terraform and we use it more often in customer projects as well as internally.

References:

- <https://pages.awscloud.com/Gartner-Magic-Quadrant-for-Infrastructure-as-a-Service-Worldwide.html>
- <https://ryaneschinger.com/blog/terraform-state-move/>
- <https://www.terraform.io/docs/commands/import.html>
- <https://www.terraform.io/docs/commands/state/mv.html>
- <https://www.terraform.io/docs/commands/refresh.html>
- <https://pragmacoders.com/blog/creating-an-ec2-instance-with-terraform>
- <https://medium.com/faun/infrastructure-as-code-deploy-an-ec2-instance-with-aws-Cloudformation-3a0afa0b832c>
- <https://aws.amazon.com/blogs/aws/new-Cloudformation-drift-detection/>
- <https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/refactor-stacks.html>
- https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/resource-import.html?icmpid=docs_cfn_console#resource-import-overview
- <https://Cloudfonaut.io/rolling-update-with-aws-Cloudformation/>
- <https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-helper-scripts-reference.html>
- <https://aws.amazon.com/cdk/>
- <https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/transform-aws-serverless.html>
- <https://docs.hashicorp.com/sentinel/>
- <https://github.com/aws-Cloudformation/cfn-python-lint>