Infrastructure as Code with focus on Terraform

Hitesh Sharma (hiteshs@kth.se), Gibson Chikafa (chikafa@kth.se)

May 31, 2020

1 Introduction

In traditional approach to infrastructure management, there were servers which had to be set-up manually by a separate team on-premise. The task involved stacking servers on the server racks, configuration of them on the requirements specific to the operating system used and application being hosted [22]. There are many challenges of this approach to infrastructure management. It requires specialized engineers for storage, network and system administration. This subsequently increases the cost of the setup. Also, maintaining backup on physical server racks and their maintenance is too much of effort. Furthermore, when scaling up in case of increased load you may need extra physical servers and when the traffic decreases those servers may become obsolete.

The emergence of cloud computing solved some of these problems. For example, organizations are not required to manage any physical hardware on premise to satisfy software operations needs. However, there is still a need to spin up cloud instances and other dependent components instantly to effectively deal with the concerns of agility, load balancing and high availability[22]. This is where Infrastructure as Code (IaC) comes in.

Infrastructure as Code (IaC) is the management of infrastructure (networks, virtual machines, load balancers, and connection topology) in a descriptive model, using the same versioning as the DevOps team uses for source code[2]. That means the core best practices of DevOps such as version control, virtualized tests, and continuous monitoring are applied to the underlying code that governs the creation and management of the infrastructure. Also, like the principle that the same source code generates the same binary, an IaC model generates the same environment every time it is applied[2]. There are many tools that have been developed to manage infrastructure as code such as Amazon CloudFormation, Azure Resource Manager, Google Cloud Deployment Manager, Puppet, Ansible, Chef and Terraform. Each tool was developed with a specific intent in mind and one tool may be a better fit than another. Choosing the right tool matters to

successfully fulfill most IaC needs and it involves understanding the important abstractions of IaC tools and use cases they are best suited for[21].

The purpose of this essay is to introduce IaC, benefits of IaC, some important properties that helps to categorise different IaC tools which in turn help in deciding the right tool for your requirement. We would then take a deeper look into one of the most popular IaC tools named Terraform and how it works in combination with other tools.

2 Benefits of IaC

2.1 Easily Reproducible Systems

The same code written for the infrastructure will produce the same environment every time it is run. It is therefore possible to effortlessly and reliably rebuild any element of the infrastructure. The ability to rebuild any part of system effortlessly removes much of the risk and fear when making changes such that new services and environments can be provisioned with little effort.[4]

2.2 Increased Site Reliability

In most of the companies they use a legacy system of running monthly scans on the infrastructure for vulnerabilities by the security or a site reliability team. These health checks cannot give 100 percent and also it might interfere with human errors while executing. These issues are tackled very well by the configuration management tools like Ansible, Chef etc.

2.3 Experimenting while having expenses in control

Cloud technologies have many benefits financially as they adopt the policy of pay as you use. In combination with IaC, resources can be easily created, destroyed, replaced, resized and moved. This is a great advantage when testing. Developers can experiment the infrastructure, for example scaling up and down the same replica of the production servers, without spending much time in repetitive manual tasks of infrastructure handling. This can be done by a limited number of DevOps or system administrators who are responsible to handle the infrastructure as it all lies inside the script. Once the task is done all the virtual instances can be destroyed within a couple of minutes hence saving the company's cost.[4]

2.4 Faster, more efficient development

By simplifying provisioning and ensuring infrastructure consistency, IaC can confidently accelerate every phase of the software delivery lifecycle. For ex-

ample, developers can quickly provision continuous integration/continuous deployment (CI/CD) environments, QA can quickly provision full-fidelity test environments and Operations can quickly provision infrastructure for security and user-acceptance testing. When the code passes testing, the application and the production infrastructure it runs on can be deployed in one step.[3]

3 Categorisation of IaC Tools

To make a better choice while choosing the IaC tool(s) for use, it is important to understand the important differences between these IaC tools. Though over the years IaC tools have evolved to meet almost all use cases and user preferences, there are still some important differences that are worth mentioning. IaC tools differ according to type: Configuration management or Provisioning; Infrastructure paradigm: Mutable or Immutable; Language: Declarative or Procedural and Cloud platform: platform specific or any platform.

3.1 Type

IaC tools can be classified into two groups according to type: Configuration management and Provisioning[1]. Configuration management tools are designed to install and manage software on existing servers. This can be the installation of packages, starting of services, placing scripts or config files on the instance, etc. Provisioning tools are designed to provision the servers themselves. By provisioning, it means the first time the infrastructure is being built. However, these categories are not mutually exclusive. Some configuration tools can for example do to some degree provisioning functions and vice versa. The focus on configuration management or provisioning refers to the original purpose the tool was built for or what kind of task the tool can do better [1].

3.2 Infrastructure paradigm

Infrastructure can be classified as mutable or immutable[3]. Mutable means further updates are installed on the existing servers and the changes will happen in-place. In immutable infrastructure every change is the deployment of a new server. A new version is created, and the old versions are destroyed. Mutable paradigm faces a problem known as configuration drift[3]. As those modifications and changes happen, the configuration of the applications and infrastructure changes. Over time the configuration installed and that currently running become different making bug detection difficult.

3.3 Language

Language of IaC tools can be classified into two groups: Procedural and Declarative[6]. Procedural approach requires the user to define the steps to take to create and configure resources while declarative allows the user to specify what the configuration should be and let the system figure out the steps to take[7].

3.4 Cloud platform

Some IaC are platform specific, especially the proprietary ones. However, many IaC tools are open source and can be used for any cloud platform. The advantage with platform agnostic tools is that you can manage a heterogeneous environment with the same workflow in a situation where you have different cloud and platforms to support various platforms.

4 Terraform: one of the leaders in IaC

Terraform is an open-source infrastructure as code tool developed and maintained by HashiCorp[16]. It was first released in 2014 as open-source with a Mozilla Public License. Though it has been around for few years, Terraform has emerged as one of the famous IaC tools used across the various companies teams for automating infrastructure. It is an all platform provisioning tool but can also be used to manage the existing infrastructure. It uses a declarative approach to code the infrastructure. It is backed by a strong growing community forum [14].

4.1 The Basics of Terraform

Terraform users define infrastructure in a language called HCL (HashiCorp Configuration Language). It is a declarative and easy to understand from the syntax. An example of HCL file is shown in listing 1.0. This will provision a new Azure resource group with name myTFResourceGroup in the Azure Microsoft US East datacenter.

```
# Configure the Azure provider
provider "azurerm" {
    version = "~>1.32.0"
}

# Create a new resource group
resource "azurerm_resource_group" "rg" {
    name     = "myTFResourceGroup"
    location = "eastus"
}
```

`Listing 1: Terraform Infrastructure HCL file`

Users generally write unique HCL configuration files or borrow existing templates from the Terraform public module registry[12].

The core Terraform workflow has generally three steps: Write, Plan and Apply[13]. This workflow can be executed in all contexts such as working in a team that is collaborating on infrastructure and using Terraform Cloud. DevOps practices can be followed in each context.

1. Write: this is actually authoring the infrastructure code using some text editor in HCL. In a dynamic team where each member is making changes to the configuration files, by working in branches, the number of input variables (e.g. API Keys, SSL Cert Pairs) required to run a plan increases[13]. To mitigate the burden and the security risk of each team member arranging all sensitive inputs it is recommended to use Continuous Integration (CI) pipeline i.e using Amazon S3 to store all sensitive inputs. Another option is to use Terraform Cloud which provides a centralized and secure location for storing input variables[15].

2. Plan: Terraform enables previewing of changes before applying. In a team, the plan output creates an opportunity for team members to review each other's work. A team member will start by creating a pull request within the version control system. The pull request can include the speculative plan generated locally and other team members can review the plan based on the attached speculative plan. Other teams configure the version control to generate the plan automatically when a pull request is made. Other members can then approve the pull request based on the generated execution plan. When using Terraform Cloud, once a pull request is done, it will automatically run the plan. In both contexts when there are no conflicts, the proposed changes can be merged to the master.

3. Apply: this is actually provisioning the new infrastructure. For teams using CI pipeline this involves carefully looking into the build log before confirming the apply. Terraform Cloud presents the concrete plan to the team for review and approval[13] after a merge. The team can then discuss any outstanding questions about the plan before the change is made.

4.2 Terraform State

The State is the Terraform's memory. It is a JSON document that is used by Terraform to map real world resources to your configuration, keep track of metadata, and to improve performance for large infrastructures[17]. When

a definition file is changed Terraform can check the state of the already created resources and decide what action to take to apply the changes. If there are major changes a restart might be needed for example. In listing 2 an example of a Terraform state file can be seen. It records all important things about the resources that have been created by Terraform, like instance id, ip address and such.


```
"aws_instance.example": {
    "type": "aws_instance",
    "primary": {
        "id": "i-66ba8957",
        "attributes": {
            "ami": "ami-2d39803a",
            "availability_zone": "us-east-1d",
            "id": "i-66ba8957",
            "instance_state": "running",
            "instance_type": "t2.micro",
            "network_interface_id": "eni-7c4fcf6e",
            "private_dns": "ip-172-31-53-99.ec2.internal",
            "private_ip": "172.31.53.99",
            "public_dns": "ec2-54-159-88-79.compute-1.amazonaws.com",
            "public_ip": "54.159.88.79",
            "subnet_id": "subnet-3b29db10"
        }
    }
}
```


Listing 2: Terraform State file

Terraform state can contain sensitive data, depending on the resources in use. The state contains resource IDs and all resource attributes. For resources such as databases, this may contain initial passwords. If any sensitive data (like database passwords, user passwords, or private keys) is managed with Terraform , the state itself should be treated as sensitive data[15]. It is recommended that storing state remotely can provide better security[15]. For example, Terraform Cloud or AWS S3 backend can be used as they both support state encryption.

5 Use cases of Terraform

Let's discuss some exciting use cases of Terraform in the following section:

5.1 Multi-Tier Applications

As we know the idea of having tiers in an IT infrastructure is necessary to scale up the complete tier in case of need. Due to this, companies practice

N-tier applications (database tier, API servers, caching servers etc.). In Terraform, each tier can be described as a collection of resources, and the dependencies between each tier are handled automatically: Terraform will ensure the database tier is available before the web servers are started and that the load balancers are aware of the web nodes. Each tier can then be scaled easily using Terraform by modifying a single count configuration value. Because the creation and provisioning of a resource is codified and automated, elastically scaling with load becomes trivial.[8]

5.2 Multi-Cloud Deployment

To facilitate adaptability to handle environment failures multi-cloud infrastructure is recommended. It facilitates fault-tolerance. Some companies uses multi-region deployment like AWS have AZ (Availability zones) to serve this purpose but by using only a single cloud provider, fault tolerance is limited by the availability of that provider. Having a multi-cloud deployment allows for more graceful recovery of in conditions of loss of a region or crashing of service provider. Realizing multi-cloud deployments can be very challenging as many existing tools for infrastructure management are cloud-specific. [9]

Terraform is cloud-agnostic and allows a single configuration to be used to manage multiple providers, and to even handle cross-cloud dependencies. This simplifies management and orchestration, helping operators build large-scale multi-cloud infrastructures.

5.3 Disposable Environments

Companies usually have more than one environment like dev, staging, QA etc. They are so far the replica of production environment. Environments like QA are essential for testing the new features of application before final release to production. Companies often face difficulty of replicating production environment because it grows gradually and becomes more complex to clone. To overcome this challenge of maintaining up-to-date staging environment we could use Terraform by codifying production environment and then sharing the same code for other low level environments. These configurations can be used to rapidly spin up new environments to test in. Once they are not in use they can be easily disposed of. Terraform can help tame the difficulty of maintaining this setup of parallel environments.[10]

5.4 Resource Schedulers

There are a number of schedulers to solve the problem of static assignment of applications to machines like Mesos, YARN, and Kubernetes. These can be used to dynamically schedule Docker containers, Hadoop, Apache Spark, and many other software tools. Resource schedulers can be treated as a provider, enabling Terraform to request resources from them. This allows

Terraform to be used in layers: to setup the physical infrastructure running the schedulers as well as provisioning onto the scheduled grid.[11]

6 Terraform in combination with other DevOps technologies

IaC tools are designed with different goals and one tool maybe a better fit than the other for a particular task. Teams often feel the need to use different tools in tandem to achieve their goals. We will look at scenarios where Terraform can be used together with some other IaC tools and how this can be achieved.

6.1 Terraform with Ansible

Terraform is best at provisioning infrastructure but it does not come with a configuration management system. Ansible is a configuration management tool[21]. In scenario where you would want to provision plus do the configuration, Terraform can be used 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. Then Ansible can be used for infrastructure configuration and application installation.

Terraform and Ansible can be integrated in different ways: Terraform invoking Ansible or Ansible invoking Terraform.

1. *Terraform invoking Ansible:* The mechanism for invoking external tools in Terraform is called a provisioner, and it includes several built-in options. The two applicable to Ansible Automation are `local-exec` and `remote-exec`. The `local-exec` provisioner, which allows any locally installed tool to be executed, can be used to invoke Ansible Automation locally on the same machine as Terraform. This provisioner is used when Ansible Automation is configuring a machine over the network. Listing 2 shows an example of Terraform configuration using the `local-exec` to provision a VM. The `remote-exec` provisioner, which allows Terraform to execute commands against a remote resource, can be used to invoke Ansible Playbooks on remote resources after creation. This provisioner is used when Ansible Automation is running on the machine being configured.[20]

```
hcl
resource \aws_instance" \web" {
    # ...
    provisioner \local-exec" {
        command = \ansible-playbook -u ubuntu -i
```

```
            '${aws_instance.web.
            public_dns},' main.yml"
        }
    }
```

Listing 2

2. *Ansible invoking Terraform:* Ansible (version latest 2.5), has a built-in Terraform module which allows one to use Terraform from within an Ansible Playbook[18]. This approach requires Terraform to be installed and available in the system path of the Ansible control node. Ansible Automation uses the locally installed Terraform binary to execute commands[20].

6.2 Terraform with Packer

For organizations that want a tool capable of making the automated creation of machine image possible, HashiCorp offers an automation tool called Packer. It embraces modern configuration management by encouraging the use automated scripts to install and configure the software within the Packer-made images[19]. Packer can be used to package apps as virtual machine images and Terraform can be used to deploy servers with these virtual machine images and the rest of the infrastructure, including the network topology (i.e., VPCs, subnets, route tables), data stores (e.g., MySQL, Redis), and load balancers.

Once an image is built with Packer and uploaded to an artifact store, the new image can be used to provision new infrastructure in Terraform. Listing 3 is a sample Terraform configuration which provisions a new AWS EC2 instance. The `aws_ami_id` is a variable which will be provided when running `terraform plan` and `terraform plan`. This variable references the latest AMI generated with the Packer build in CI/CD.

```
variable "aws_ami_id" { }

provider "aws" {
  region = "us-west-2"
}

resource "aws_instance" "web" {
  ami           = "${var.aws_ami_id}"
  instance_type = "t2.micro"
}
```

6.3 Terraform with Kubernetes

Terraform offers an effective way to manage both compute for your Kubernetes cluster and Kubernetes resources. As we know Kubernetes is an open-source workload scheduler with focus on containerized applications. There are at least 2 steps involved in scheduling a container on a Kubernetes cluster. We need the Kubernetes cluster with all its components running somewhere and then schedule the Kubernetes resources, like Pods, Replication Controllers, Services etc.[5]. While you could use kubectl or similar CLI-based tools mapped to API calls to manage all Kubernetes resources described in YAML files, orchestration with Terraform presents a few benefits as described bellow:

1. Use the same configuration language to provision the Kubernetes infrastructure and to deploy applications into it.

2. Drift detection: Terraform plan will always present you the difference between reality at a given time and config you intend to apply.

3. Synchronous feedback: While asynchronous behaviour is often useful, sometimes it's counter-productive as the job of identifying operation result (failures or details of created resource) is left to the user. e.g. you don't have IP/hostname of load balancer until it has finished provisioning, hence you can't create any DNS record pointing to it.

4. Full lifecycle management: Terraform doesn't just initially create resources, but offers a single command for creation, update, and deletion of tracked resources without needing to inspect the API to identify those resources.

We have just looked at few cases how these tools can be used together in conjunction to do more sophisticated tasks. There are however pitfalls with these approaches. Firstly, there are complexity issues since using multiple tools will undeniably require more knowledge from the developers that setup and maintain such a work flow. Secondly, is the cost since not all tools are free. For smaller operations, going with the tool that best satisfies the biggest need would be a more viable option.

7 Conclusion

Infrastructure as Code is the modern way of managing infrastructure. Many IT companies can now implement stable and reliable environments quickly and at scale. Any manual intervention can be avoided and consistency could be enforced by the representation of the desired state of infrastructure in the infrastructure script. Terraform is one of the most popular IaC tools

because it is open-source, immutable, declarative and cloud-agnostic. Terraform is quite beneficial for provisioning infrastructure, therefore larger organizations with strong provisioning and configuration needs picks different combinations of Terraform with other tools as shown in the examples in this essay.

References

[1] *Why we use Terraform and not Chef, Puppet, Ansible, SaltStack, or CloudFormation*, 2016 (accessed April 26, 2020). https://blog.gruntwork.io/why-we-use-terraform-and-not-chef-puppet-ansible-saltstack-or-cloudformation-7989dad2865c.

[2] *What is Infrastructure as Code?*, 2017 (accessed May 19, 2020). https://docs.microsoft.com/en-us/azure/devops/learn/what-is-infrastructure-as-code.

[3] *Infrastructure as Code (IaC)*, 2019 (accessed April 27, 2020). https://www.ibm.com/cloud/learn/infrastructure-as-code.

[4] *infrastructure-as-code by IBM*, 2019 (accessed April 30, 2020). https://www.ibm.com/cloud/learn/infrastructure-as-code.

[5] *Terraform and Kubernetes together*, 2019 (accessed May 16, 2020). https://www.terraform.io/docs/providers/kubernetes/guides/getting-started.html.

[6] *Ansible vs Terraform vs Puppet: Which to Choose?*, 2020 (accessed April 26, 2020). https://phoenixnap.com/blog/ansible-vs-terraform-vs-puppet.

[7] *Cloud Deployment Manager — Google Cloud*, 2020 (accessed April 26, 2020). https://cloud.google.com/deployment-manager.

[8] *Use Cases - Terraform By Harshcop*, 2020 (accessed May 16, 2020). https://www.terraform.io/intro/use-cases.html#multi-tier-applications.

[9] *Use Cases - Terraform By Harshcop*, 2020 (accessed May 16, 2020). https://www.terraform.io/intro/use-cases.html#multi-cloud-deployment.

[10] *Use Cases - Terraform By Harshcop*, 2020 (accessed May 16, 2020). https://www.terraform.io/intro/use-cases.html#disposable-environments.

[11] *Use Cases - Terraform By Harshcop*, 2020 (accessed May 16, 2020). https://www.terraform.io/intro/use-cases.html#resource-schedulers.

[12] *Browse Modules — Terraform Registry*, 2020 (accessed May 19, 2020). https://registry.terraform.io/browse/modules.

[13] *The Core Terraform Workflow*, 2020 (accessed May 19, 2020). https://www.terraform.io/guides/core-workflow.html].

[14] *Latest Terraform Topics - Harshcop Discuss*, 2020 (accessed May 19, 2020). https://discuss.hashicorp.com/c/terraform-core.

[15] *Sensitive Data in State*, 2020 (accessed May 19, 2020). https://www.terraform.io/docs/state/sensitive-data.html.

[16] *Terraform(Software)*, 2020 (accessed May 19, 2020). https://en.wikipedia.org/wiki/Terraform_(software).

[17] *What is Infrastructure as Code?*, 2020 (accessed May 19, 2020). https://www.terraform.io/docs/state/index.html.

[18] *Ansible and HashiCorp: Better Together*, 2020 (accessed May 20, 2020). https://www.hashicorp.com/resources/ansible-terraform-better-together/.

[19] *Harshcop Packer: Build Automated Machine Images*, 2020 (accessed May 20, 2020). https://www.packer.io/.

[20] *HASHICORP TERRAFORM AND RED HAT ANSIBLE AUTOMATION*, 2020 (accessed May 20, 2020). https://www.redhat.com/cms/managed-files/pa-terraform-and-ansible-overview-f14774wg-201811-en.pdf.

[21] *Infrastructure as Code: Chef, Ansible, Puppet, or Terraform*, 2020 (accessed May 20, 2020). https://www.ibm.com/cloud/blog/chef-ansible-puppet-terraform.

[22] Kief Moris. *Infrastructure as Code: Managing servers in the cloud.* O'Reilly, 2016.