Introduction to Infrastructure as Code in a Multi-Cloud Environment

From physical servers to virtual machines to spinning up cloud resources, setting up your infrastructure has always been in the domain of the operations team. While trying to keep the existing environments running and secure, they are often tasked with manually setting up new or updating environments for the development staff. The manual process is not only time consuming, but can lend to human error, and a potential lack of rigor when setting up the environment. Infrastructure as code (IaC) aims to make the process of creating infrastructure repeatable, trackable and auditable. This process allows system administrators to define the resources they need to provision with code, ranging from network configuration, to resources and data storage in your cloud environment. By utilizing code to create the environment, you can ensure that the infrastructure has been created as specified while providing a repeatable process to set up exact replicas in all phases of your SDLC (dev, stage, and production). Since code is used, it can (and should) be checked into a source control repository such as GIT, which will allow you to quickly see the current state of the environment, as well as provide the ability to fork the configuration to test changes to the environment. This source-controlled infrastructure code will also provide a detailed audit trail for any changes to the environment and can also serve as “documentation” in case your sys admin wins the lottery and decides to move to Hawaii.

How do I start?

While the time to set up the infrastructure as code ecosystem (and training) may not be practical if you only have a few servers or a small on-prem environment, as you start to transition to a cloud-first environment, leveraging IaC could benefit your organization. All the major cloud providers have their own version of infrastructure as code, and use different languages and syntax to render the code:

<table>
<thead>
<tr>
<th>Provider</th>
<th>IAC</th>
<th>Supported Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Cloud Platform</td>
<td>Deployment Manager</td>
<td>YAML</td>
</tr>
<tr>
<td>Amazon Web Services</td>
<td>Cloud Formation</td>
<td>YAML or JSON</td>
</tr>
<tr>
<td>Microsoft Azure</td>
<td>Azure Resource Manager</td>
<td>JSON</td>
</tr>
</tbody>
</table>

If you have a single cloud environment, learning one of the provider specific platforms makes sense, as the code will be specific to that environment. However, in the multi-cloud world, having to learn multiple languages and syntax will be an additional burden on the sys admin staff. Thankfully, HashiCorp has created Terraform, which is an open source tool that allows you to run IaC on all the main cloud providers, with consistent, readable syntax. This uses HashiCorp’s Command Interface language, which is simple and very human readable (especially for anyone that has spent hours debugging JSON for that missing bracket). Terraform takes declarative configuration files and the concept of “providers” to interact with the various cloud systems via API to create the desired resources. Terraform also allows you to map resource dependencies so you can understand how a minor infrastructure change could have a potential cascading effect across your environment. Utilizing the Terraform scripts will also allow you to ensure that the same environmental configurations are in dev, test, staging and production, to help promote your application through the process quicker, without having to hear “it worked in dev!”.
What does it look like?
As mentioned above, Terraform uses HCI syntax to create .TF files. Terraform is run on a folder in your file structure, and all files with the .TF extension are analyzed, dependencies are resolved, and a build plan is created for that specific state. You can separate resources into different Terraform files, so that you can re-use parts (e.g. build resources on another environment that already has a network defined). The examples below show TF files across the main three providers (Amazon, Google, and Microsoft) to create a micro compute resources (VM) with Ubuntu OS.

The red boxes signify the provider declaration. The green boxes define the resource type, and the yellow boxes define the machine type and OS image. As you can see, with a few lines of code, you can quickly and repeatably stand up resources in your cloud environment. There are also many other parameters you can add to each resource listing to customize it as necessary (such as disk size, service accounts, network interfaces, etc.). Hashicorp has guides for AWS, GCP, and Azure, that provide information and examples for each provider.

How do I run it?
If you are logged into your cloud environment, most services provide a cloud shell interface that already has Terraform installed (see example below from Google Cloud Platform).
After you navigate to the directory where your Terraform files are, the first step is to run `terraform init`. This command scans the code in that folder to understand what to build, what the dependencies are, and to download the plugins for the provider.

```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to  .
Use "gcloud config set project [PROJECT ID]" to change to a different project.
$ terraform --version
Terraform v0.12.2
```

The next step is to run `terraform plan`, which is a dry-run to show you what changes would be made to your infrastructure. Terraform also checks against a state file (terraform.tfstate) that it creates or updates every time it is run to compare the desired state to the as-is or known state. If the infrastructure does not match the desired state then Terraform will compute the difference, and suggest bringing it back in line by reverting the manual changes (e.g. someone adds an unapproved firewall rule, or changes a resource tag).
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be persisted to local or remote state storage.

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

```hcl
# google compute instance.default will be created
  resource "google_compute_instance" "default" {  
    can_ip_forward = false  
    cpus           = (known after apply)  
    deletion_protection = false  
    guest_accelerator = (known after apply)  
    id              = (known after apply)  
    instance_id     = (known after apply)  
    label_fingerprint = (known after apply)  
    machine_type    = "f1-micro"  
    metadata_fingerprint = (known after apply)  
    name            = "hello-world-gcp"  
    project         =  
    self_link       = (known after apply)  
    ssd_fingerprint = (known after apply)  
    zone            = "us-central1-a"  
    boot_disk {  
      auto_delete = true  
      device_name = (known after apply)  
      disk_encryption_key_sha256 = (known after apply)  
      source = (known after apply)  
      initialize_params {  
        image = "ubuntu-on-cloud/ubuntu-1804"  
        size = (known after apply)  
        type = (known after apply)  
      }  
    }  
    network_interface {  
      address = (known after apply)  
      name = (known after apply)  
      network = "default"  
      network_ip = (known after apply)  
      subnetnetwork = (known after apply)  
      subnetnetwork_project = (known after apply)  
      access_config {  
        assigned_net_ip = (known after apply)  
        net_ip = (known after apply)  
        network_tier = (known after apply)  
      }  
    }  
    scheduling {  
      automatic_restart = (known after apply)  
      on_host_maintenance = (known after apply)  
      preemptible = (known after apply)  
      node_affinities {  
        key = (known after apply)  
        operator = (known after apply)  
        values = (known after apply)  
      }  
    }  
  }  

Plan: 1 to add, 0 to change, 0 to destroy.
```
The output of the `terraform plan` command above delineates the resources and the configurations that will be created when Terraform runs. The output above shows that a new Google compute instance will be created. This is a very useful tool to validate changes to your environment before they are made and potentially cause unintended consequences. If the plan looks good, the next step is to run `terraform apply` to invoke the changes by calling the provider APIs.
The `terraform apply` command will re-iterate the configuration described in the `plan` step and will also ask for a confirmation that you want to take these actions. After a few seconds, the resource is provisioned, and you can verify in your cloud console.

To remove resources or configurations, you can run the `terraform destroy` command. The output will show the changes in settings, and similarly to the `apply` command, this will require confirmation before the resources are removed.

```
$ terraform destroy
```

An execution plan has been generated and is shown below. Resource actions are indicated with the following symbols:

- destroy

Terraform will perform the following actions:

```
# google_compute_instance.default will be destroyed
resource "google_compute_instance" "default" {  
  - can_ip_forward  = false  -> null  
  - cpu_platform    = "Intel Haswell"  -> null  
  - deletion_protection = false  -> null  
  - guest_accelerator = []  -> null  
  - id              = "helloworld-gcp"  -> null  
  - instance_id     = "3851499846057056934"  -> null  
  - label_fingerprint = "422N5pB5rSH-"  -> null  
  - labels          = []  -> null  
  - machine_type    = "f1-micro"  -> null  
  - metadata        = {}  -> null  
  - metadata_fingerprint = "p61jTvISj4g-"  -> null  
  - name            = "helloworld-gcp"  -> null
```
About OnPoint


OnPoint is a part of the Publicis Sapient platform, with access to industry leading AI tools and teams. Contact us at innovation@onpointcorp.com or visit onpointcorp.com to learn more about us and our services.