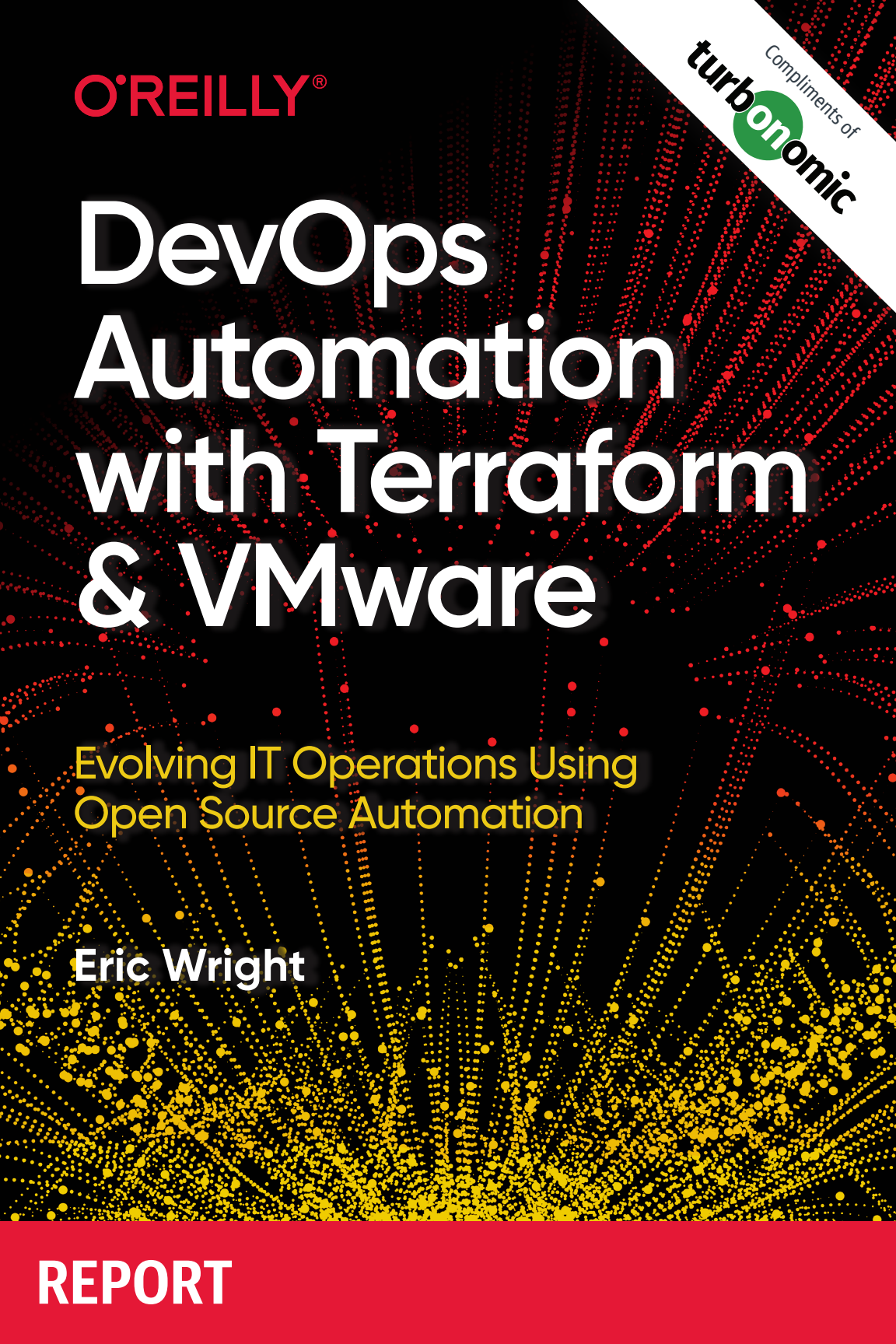
O'REILLY®
Compliments of
turbonomic
DevOps Automation with Terraform & VMware
Evolving IT Operations Using Open Source Automation
Eric Wright
REPORT

turbonomic

APPLICATION RESOURCE MANAGEMENT

Bridge the gap between applications and infrastructure, eliminating application performance risk without overprovisioning infrastructure.

Learn More: bit.ly/turbo-devops

Trusted by the world's largest organizations, including more than 100 of the Fortune 500

DevOps Automation with Terraform and VMware

Evolving IT Operations Using Open Source Automation

Eric Wright

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

DevOps Automation with Terraform and VMware
by Eric Wright

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Nikki McDonald
Developmental Editors: John Devins and Amelia Blevins
Production Editor: Kristen Brown
Copyeditor: Octal Publishing, LLC
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

August 2019: First Edition

Revision History for the First Edition
2019-08-01: First Release

978-1-492-07373-4

[LSI]

Table of Contents

Foreword

The world of software infrastructure has evolved dramatically in the past few years. In a span of 20 years, we made the shift from physical machines to virtualization, which simplified management and allowed us to make changes faster while improving our hardware utilization. In 2006, Amazon Web Services (AWS) introduced Amazon Elastic Compute Cloud (Amazon EC2) and officially started the cloud era. Today, we have multiple public cloud vendors, and companies from one-person startups to the largest Fortune 100 organizations are all adopting public cloud.

Terraform was created to enable users to easily consume cloud infrastructure, following an Infrastructure as Code (IaC) approach. By codifying our infrastructure, we get up-to-date documentation, simple version control, peer reviews, rollbacks, and, most important, automated provisioning. Terraform is made extensible through a provider plug-in model that allows it to manage low-level hardware, Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) all at once. Modern infrastructure is often composed of all these layers being used together and Terraform allows for a consistent provisioning workflow across all our infrastructure.

As developers consume more cloud services and we push to iterate faster, Terraform provides a solution for managing our infrastructure and making provision simple. We are glad to see an ever-growing community of users and hope that this report allows users to gain more familiarity and jump into managing infrastructure with Terraform.

— Armon Dadgar, cofounder and CTO of HashiCorp

Having run datacenters at scale in some challenging environments, the value of codifying infrastructure builds became very apparent to me early on. Initial work with PowerCLI and Puppet proved to be helpful; however, the arrival of Terraform changed the game when it comes to building and operating VMware infrastructure. Add in hybrid environments that reach into one or more public cloud environments and you will quickly realize the complexity in front of you as an architect or developer.

Being able to embrace using code to define and manage infrastructure and then sharing that code to rapidly enable development teams to consume and create their own resources will be the first in your steps toward more DevOps-oriented infrastructure management. Welcome to the journey toward IaC, and I hope that this report helps you along your path.

— Eric Wright, technology evangelist, Turbonomic

DevOps Automation with Terraform and VMware

DevOps practices are widely adopted and proven in many of the most challenging as well as some of the most seemingly simple environments. The reason this is important to you is that no matter what the size or scale of your environment, you can gain value by implementing DevOps methods in your teams. DevOps is as much a people and process change as it is a technology change, if not more so on the people and secondarily on process.

You will come away from this guide with an understanding of how DevOps processes and IaC can apply to your VMware infrastructure using Terraform, beginning with building virtual machines (VMs) using the Terraform command-line interface (CLI), and further by gaining knowledge about how to use Terraform and VMware to operate your infrastructure. All of the concepts are explained as you work through the guide, including specific examples and links to some more complex examples in the companion code repository.

Before we begin, let's explore what Terraform is and why it is important to you. Terraform is described by HashiCorp as an open source tool to write, plan, and create IaC. Terraform is lightweight and versatile, with the ability to use numerous modules to simply create code using declarative configuration for many different components across any hybrid infrastructure.

This guide uses the Terraform OSS, which is a freely available open source tool. Terraform Enterprise is a commercial version that includes more options with workspaces, team collaboration features, private module registries, security, self-service options, and more. The concepts discussed here also apply to the Enterprise edition.

How Do You Manage Your Infrastructure Today?

It's good to do a health check on how you operate your infrastructure today to map against what you will learn in this guide. Which one of these statements would you use to describe your current processes?

1. Manually build, deploy, and patch VMs
2. Deploy VM templates from templates and patch using remote batch processes or remotely connecting to servers
3. Deploy VMs from template and patch remotely from a central update source (e.g., SCCM)
4. VMs are deployed and replaced when patches are implemented

How you build and manage your applications is also important to understand. Which of the following statements would describe your current application deployment process?

1. Manually deploy application code and packages from files/source on servers via console or remove SSH/WinRM/RDP
2. Manually deploy application code and packages from code repositories via console or remove SSH/WinRM/RDP
3. Manually deploy application code and packages from a central server/workstation
4. Automatically deploy application code and packages from a central source environment during specific schedules
5. Automatically deploy application code and packages from a central source environment regularly when any code updates are committed (aka Continuous Deployment)

If you are using mostly manual processes today, the goal of this guide is to introduce you to automation. Teams that chose the middle or higher options from the two lists will learn how Terraform maps to your current processes. If you're already using DevOps processes in other parts of your infrastructure (e.g., cloud, containers) this guide will help to bring some of those practices to your existing virtualization environment.

CHAPTER 1

Infrastructure as Code and Immutable Infrastructure Concepts

In this chapter, we explore concepts of operating an environment using Infrastructure as Code (IaC). You learn about *immutable infrastructure* and how to apply it to operational practices within your own VMware environment. We begin with a concept overview and learn the Terraform terminology. Next, we explore the Terraform vSphere provider and offer operational tips and guidance for your Terraform environment.

It is assumed that you have a version control system. Examples in this guide use Git with code hosted on GitHub.com in public repositories, which will be updated based on feedback and questions from readers.

Terraform Terminology and Component Descriptions

Let's begin by defining some key phrases and terminology that you will encounter when using Terraform. These terms will show up throughout the guide and in the online documentation, and give context to what you are building in the example scenarios that we present.

Provider
: This is your connector to the underlying infrastructure. This is how your declarative code will interact with the management API of whichever platform you are building on. The VMware vSphere provider will communicate with either vCenter for a full coverage or to a vSphere host, which provides less functionality.

Input variables
: Input variables are passed to your Terraform configuration. These can be dynamically created or statically assigned. If not assigned programmatically, you will be prompted at the command-line interface (CLI) to enter values.

Data sources
: These can be computed or queried infrastructure sources that are used in other parts of your Terraform configuration, such as clusters, resource pools, regions, or any of a variety of objects. Data sources will vary based on which provider you are using (e.g., Amazon Web Services [AWS], VMware, Digital Rebar).

Expressions
: These are computed results that can range from literals to variables to indices, maps, and many other types. It's possible to have queries that can also feed other expressions and do things like count within resources and then dynamically assign the count to naming resources. Another example is creating a virtual machine (VM) and then assigning the network interface on creation to a virtual switch or dynamically assigning VM names based on count, or environment name.

Functions
: These are built-in functions that you can use in your expressions, including numeric (e.g., `min`, `max`), string (e.g., `lower`, `upper`, `substr`), collection (contains, flatten, sort, merge), and many others. You can find a full set of built-in functions here.

Output values
: Return values from your Terraform environment. These can include static or dynamic results such as VM name, IP address, storage location, and other computed results that are available after a Terraform resource deployment or update. These are

available programmatically from the CLI as soon as you have run your Terraform configuration.

Why DevOps and IaC?

Let's begin by clearly describing what we mean by DevOps and IaC. There are many versions of the definition of DevOps. The often-referenced Wikipedia article describes it as the following:

> DevOps is a set of software development practices that combines software development (*Dev*) and information technology operations (*Ops*) to shorten the systems development life cycle while delivering features, fixes, and updates frequently in close alignment with business objectives.

Similarly, the IaC definition is listed on Wikipedia as follows:

> Infrastructure as code (IaC) is the process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.

The definition is less important than understanding the goal of these practices, which is to build and deploy infrastructure and applications in a consistent way, faster, and with less risk. It really is that simple.

Version control systems allow you to store, tag, and version your code centrally. The common "source of truth" of your code is then cloned or forked to the target client and servers, which ensures consistency. As each code and application update is stored, you can test it across environments (e.g., Dev, QA, Test, Production).

Using consistent build processes stored in code, in a collaborative way, ensures consistency of outcome and faster time to deploy. Using simple delegation of privileges for the application development teams allows the same deployment processes to be used across teams and across environments. Building application infrastructure will now be truly on-demand and consistent without the lengthy wait times to submit requests and hand tasks back and forth between teams.

Immutable Concepts and Capabilities with Terraform

Immutable infrastructure is a big shift from the traditional practice of build, deploy, patch, and maintain, which is the common method used in most data centers and even in cloud infrastructure today. Immutable infrastructure is built and deployed, and then designed to not be changed after deployment. The growth in features and popularity of containerized infrastructure has made immutable infrastructure practices simpler. VMs can be deployed in a similar way, provided that the application is built with immutable infrastructure in mind. The advantage to immutable infrastructure is the speed and stability when building and deploying applications, thanks to consistency of the underlying infrastructure layers.

Terraform allows for the rapid build, deployment, and tear down of infrastructure and applications. Figure 1-1 illustrates the flow as applications are developed and then built and deployed. Teams can choose to use packaging or build-from-source with repeatable deployment of the underlying infrastructure and the applications to ensure consistency across all environments.

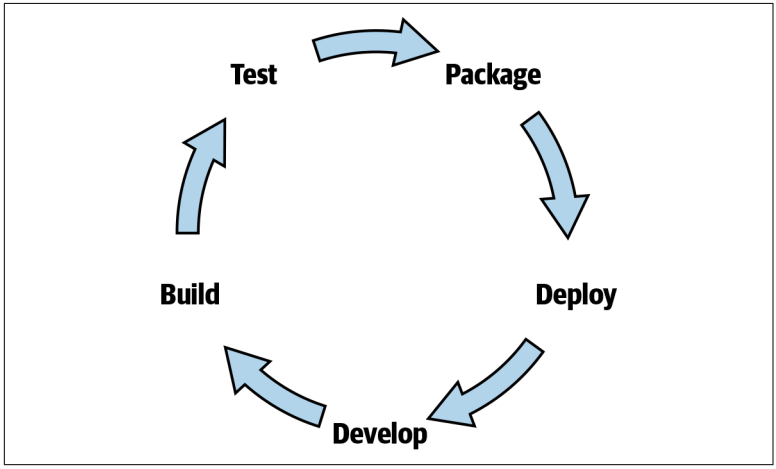

Figure 1-1. Build and deployment cycle

Terraform Features of the VMware vSphere Provider

Each Terraform provider includes a number of data sources and parameters that you will use to configure your infrastructure. Each resource will have some minimum required values (e.g., name, cluster, host) and many optional parameters.

Should I Make Everything into IaC?

You probably have applications that receive limited updates, and it does not make sense to invest huge time and effort to codify into declarative resources. Use your judgement on whether each application or environment is likely to gain value from repeatable code-powered processes.

You can find configuration parameters and documentation for each data source on the Terraform website for the most updated version. Terraform is under continuous development and is rapidly expanding coverage. You can review information and check for updates in future versions for VMware providers and a growing list of other providers here.

Operational Practices for Terraform

Embracing IaC and DevOps methodologies also means changing some of your processes to adapt to this style of infrastructure management. This is a journey toward more agile, rapid, and consistent deployment of application resources with many stages of adoption. Don't feel that you are losing the battle because you are not running 200 deploys a day like a LinkedIn or a Pinterest team would. Your goal is to take advantage of the power of Terraform and your existing VMware environment to move in the direction of IaC. As the proverb goes, "A journey of a thousand miles begins with a single step."

Using Environment Variables for Configuration Parameters

Building and deploying applications and VMs with static server and location references is a risky practice. There is a chance that you will

accidentally store content in the code repository, which will bleed into different environments when you deploy.

Storing configuration parameters as environment variables locally and on the remote application servers is particularly important with immutable infrastructure. As environments are spun up, they get configuration in memory. When changes are needed, modifications are also done in memory, which is quick and repeatable.

Simply define an environment variable such as `TF_VAR_yourvariablename`, as shown in Figure 1-2, and then you can refer to it anywhere in your Terraform configuration as `${var.yourvariablename}` for easy access and to remove the need for static parameters and constants defined in your code. Each server/instance can have its own environment to ensure that your configuration is localized and also dynamically created and modified.

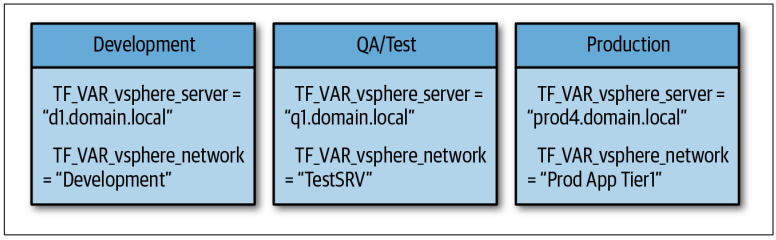

Figure 1-2. Variables defined by environment type

Storing Secrets

This is the most contentious and challenging area when it comes to infrastructure operations. Where is the best place and product to store secrets (e.g., passwords, API keys, administrative network port information, Secure Shell [SSH] keys) for use in IaC?

Like the files and folders, the answer is, "It depends." It's ideal to use a secured, programmatically accessible secret storage platform (e.g., HashiCorp Vault, CyberArk, AWS Secrets Manager). The examples in this guide use locally stored credentials and secrets using environment variables on each system.

It's critical that no secrets make their way into your code and into the repository. Even if you remove temporary passwords from code that has been previously committed to a repository, those previous

versions can be searched and viewed by anyone with access to the code repository.

Terraform Process Flow

You should have a standard flow that you follow when using Terraform. Your flow will be to create a configuration, validate the code, check the live environment, run the configuration, and then check and potentially tear down the infrastructure later on. This directly relates to the common Terraform commands, as shown in Figure 1-3. Think of this as a resource life cycle.

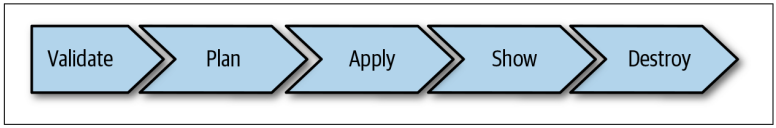

Figure 1-3. Terraform process flow

There are other commands such as `taint`, `untaint`, `refresh`, and `graph`, which we touch on later in the guide, but these core commands will be the ones you encounter the most.

Why Terraform for VMware?

The ability to create declarative configurations without having to know and test all of the different APIs is significant. This also means that you might not need to build and understand complex scripts with multiple scripting languages that have been needed for automating VMware deployments up to now.

Terraform OSS is free and extensible for your other environments (e.g., Kubernetes, AWS, Microsoft Azure, Google Cloud Platform) with limited change required. Just swap out the provider and some configuration parameters, and you have the same declarative code for the rest of your hybrid environment.

Now that you have learned the fundamentals of IaC concepts and the core knowledge of your Terraform platform, it's time to move on to working examples of the core features and functions of Terraform and the VMware vSphere. You can adapt these upcoming examples to your own local environment. They provide the foundation to begin applying these new concepts in your infrastructure and operations processes.

CHAPTER 2

Deploying and Configuring Virtual Machines

This chapter introduces our first working example of Terraform for your vSphere environment. You learn how to create declarative configurations to deploy and manage the life cycle of virtual machines (VMs), and VM snapshots.

Using Terraform alone does not create DevOps processes. The tool simply allows you to modify your existing processes to adopt more DevOps methods and processes.

Terraform, as it is used here, is more for life cycle management and continuous and repeated uses.

Configuring Your Terraform Environment

Your environment requires a minimal set of configurations to attach to and manage your VMware infrastructure. This includes configuring your VMware provider, environment variables, and data sources that will be used during Terraform-controlled resource management.

Following are example snippets of the code required. The full code samples for these scenarios are available here.

Setting Up Your Variables

Variable definition assigns a type to variables you will use, which can be simple variables (`string`, `integer`, `bool`) or complex variables (`list`, `set`, `map`, `object`, `tuple`) for more advanced scenarios. Descriptions are optional but always helpful for others who will be reading and collaborating with your code. You must define variables prior to having values assigned in your Terraform resource configuration:

```
variable "vsphere_user" {
    type = "string"
    description = "vCenter/vSphere user"
}
variable "vsphere_password" {
    type = "string"
    description = "Password for vCenter/vSphere user"
}
variable "vsphere_server" {
    type = "string"
    description = "vCenter server or vSphere host name"
}
```

This example shows the three mandatory variables required for Terraform to authenticate to the vSphere or vCenter in order to interact with your virtualization environment. You can configure other custom or optional variables similarly for use anywhere else in your Terraform configuration code.

Setting Up Your VMware Provider

There are three mandatory parameters needed for your VMware provider: the vCenter or vSphere host (IP or DNS name), a username, and password combination to authenticate with. Note the `allow_unverified_ssl`, which is also required if you are using a self-signed certificate or have not assigned a Secure Sockets Layer (SSL)/Transport Layer Security (TLS) certificate to your hosts or vCenter.

In the example that follows, notice that we assign each parameter to a variable. The contents of each variable will be requested at the command line if it is not defined but should be assigned in your workstation or server instance as environment variables using `TF_VAR_variablename` as the format:

```
 provider "vsphere" {
  user           = "${var.vsphere_user}"
  password       = "${var.vsphere_password}"
  vsphere_server = "${var.vsphere_server}"

  # If you have a self-signed cert
  allow_unverified_ssl = true
}
```

Setting Up Your Data Sources

Data sources are used for discovering resources and resource identifiers. These can include your datacenter, clusters, and resource pools. This ensures that you don't need to statically assign this information into your code so that you can reuse these configurations for multiple resources and environments.

Resource configurations can refer to these data sources as `${data.resourcename.parameter}`, where the `resourcename` and `parameter` are going to be dynamic based on your specific environment:

```
data "vsphere_datacenter" "dc" {
    name            = "${var.vsphere_datacenter}"
}

data "vsphere_datastore" "datastore" {
    name            = "${var.vsphere_datastore}"
    datacenter_id   = "${data.vsphere_datacenter.dc.id}"
}
```

The `vsphere_datacenter` pulls the name from your `TF_VAR_vsphere_datacenter` variable. The `vsphere_datastore` is then pulled from your `TF_VAR_vsphere_datastore` variable and also programmatically queries the `data.vsphere_datacenter.dc.id` to get the UUID, which Terraform needs to ensure that it uses a unique identifier.

Building a Simple Virtual Machine Resource from a Template

Now that you've set up your provider, variables, and data sources, it's time to create your VM resource. The declarative resource for a simple VM begins with a few defining criteria:

- Name of your VM

- Memory allocation
- CPU count
- Storage allocation and location
- Template to use for cloning
- Network attachment and definition

Additional options can include Resource Pools, Tags, Clusters, Datacenter, and others. Here is the basic resource configuration for your simple VM:

```
resource "vsphere_virtual_machine" "vm" {
  count            = 1
  name             = "terraform-test${count.index}"
  resource_pool_id  = "${data.vsphere_resource_pool.pool.id}"
  datastore_id     = "${data.vsphere_datastore.datastore.id}"
    num_cpus = 2
    memory   = 2048
    guest_id = "other3xLinux64Guest"

    network_interface {
      network_id = "${data.vsphere_network.network.id}"
    }

    disk {
      label = "disk0"
      size  = 20
    }

    clone {
      template_uuid="$(data.vsphere_virtual_machine.template.id}"
    }
}
```

This example shows a VM with two vCPU machines, 2 GB of virtual memory, a 20 GB primary virtual disk, and will clone from a template defined by your data sources. This one uses a Linux machine, as shown by the `guest_id = "other3xLinux64Guest"`, which defines the OS and what type of VMware Tools that vCenter will use to control it.

You are now ready to follow the life cycle flow illustrated earlier in Figure 1-3. First you run the `terraform validate` command to ensure that our syntax is correct:

```
$ terraform validate
Success! The configuration is valid.
```

Now you are ready to run the `terraform plan` command to see what Terraform will do using the configuration you've created:

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
…
… (PARTIAL OUTPUT SHOWN)

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:

  # vsphere_virtual_machine.vm[0] will be created
  + resource "vsphere_virtual_machine" "vm" {
      + boot_retry_delay                      = 10000

Plan: 1 to add, 0 to change, 0 to destroy.
```

Your plan shows that a single VM will be created and the output will show all of the configuration to be applied including some sections that are static and some that are listed as `(known after apply)` because they are dynamic values that are generated during the resource creation.

Now you can run the `terraform apply` command to go ahead with the creation of your VM:

```
vsphere_virtual_machine.vm[0]: Creating...
vsphere_virtual_machine.vm[0]: Creation complete after 9s
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

virtual_machine_default_ips = [
  "terraform-test0", 10.10.45.127
]
```

You have a running VM based on your template, which deployed in a matter of seconds. This greatly reduces the time spent doing these tasks and ensures consistency of outcome. You can also use the same code with other teams (QA, Development, etc.), which now extends the same process to set up their own environments without needing to have them navigate the VMware ecosystem at all.

Modifying Your VM Resource

Updating and modifying your resources is as simple as changing the configuration files and then running a `terraform plan` to see the resulting change followed by a `terraform apply` to enact the change. It's important to note that actions requiring a reboot will trigger the reboot automatically without prompting. Hot add hardware can be done dynamically, such as the act of increasing virtual memory.

Your first VM needs to increase memory, so you will update the configuration and run the plan. You will see the change in the output:

```
…
~ memory                                    = 2048 -> 4096
…
Plan: 0 to add, 1 to change, 0 to destroy.
```

Now you run the `terraform apply` command and initiate the changes, which modifies only the VM rather than re-creating it from scratch:

```
vsphere_virtual_machine.vm[0]: Modifying...
vsphere_virtual_machine.vm[0]: Modifications complete after 7s

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

You can see by the limited configuration code and the speed of the process to update how this flexibility helps with the already reduced amount of effort in creating your resources.

Destroying the Virtual Machine Resource

You remove your VMs at the end of the life cycle by using the `terraform destroy` command. This ensures easy removal and a complete and programmatic cleanup of all of the resources defined in your configuration. You are prompted to confirm the removal:

```
$ terraform destroy

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure,
  as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes
```

```
vsphere_virtual_machine.vm[0]: Destroying...
vsphere_virtual_machine.vm[0]: Destruction complete after 3s

Destroy complete! Resources: 1 destroyed.
```

Note that the `destroy` command will remove *everything* from your resource configuration, so you must take care as you complete the life cycle process with the destroy process to be sure that you no longer need the contents. Luckily, you can quickly and simply recreate everything by using the `terraform apply` command again.

Running Tasks and Scripts after VM Deployment

Remember that we want to aim for immutability. The use case we are building here is to deploy from a template and then run tasks and scripts, which might be an application deployment or some in-guest configurations.

There are a few ways to inject post-processing scripts into your Terraform configuration. These include inline (directly in your configuration file) or using a local file, which is sent to the target for execution.

Running scripts and commands remotely on the target system requires a remote management protocol like SSH or Windows Remote Management (WinRM) as well as any associated credentials and private keys. It's also assumed that you have network access to the resources when they are launched. Your VMware configurations are less likely to have issues with connectivity than a cloud configuration that might need private networks and virtual private networks (VPNs).

Inline Script Example

Your configuration file simply needs a code block using the `remote-exec` provisioner, as shown here:

```
provisioner "remote-exec" {
    inline = [
      "puppet apply",
    ]
connection {
        type = "ssh"
        host = "terraform-test${count.index}"
```

```
        user = "${var.photon_user}"
        password = "${var.photon_password}"
    }
  }
```

Each command will be run as the credentials provided. You can see that inline commands are run just as you would from the command line within the machine itself. This simplified example showed running a `puppet apply` command, which can register the machine to a remote system or run other VM-specific configurations for your site.

Script from File Example

You might want to use file-based scripts rather than coding your scripts within the Terraform configuration. More complex scripting is best handled outside of your Terraform configuration and it may be wise to share script files centrally so that they can be called by Terraform but versioned and updated outside of your Terraform configuration.

The following example shows a local file named *script.sh* that will be uploaded to the remote machine using SSH with credentials pulled from environment variables on the local machine running Terraform:

```
provisioner "file" {
    source      = "script.sh"
    destination = "/tmp/script.sh"

    connection {
        type = "ssh"
        host = "terraform-test${count.index}"
        user = "${var.photon_user}"
        password = "${var.photon_password}"
    }
  }
```

The *script.sh* file will only be uploaded to your remote system with this example. To run the file after it's uploaded, you simply combine these provisioners to upload the file and then to execute the file on the remote system by using the `remote-exec` option.

Controlling and Using Snapshots

Part of your more DevOps style of VM operations should include snapshot management. Cloning machines and protecting state during iterative changes with snapshots is a handy way to give you a safe and quick rollback to a previous state of the VM and the related application configuration.

Snapshots are also something that can get out of control and do have an impact on performance and storage space. There is no shortage of warnings from both HashiCorp and VMware on the use of snapshots with Terraform, and for good reason. Your use case here is to take an existing machine and add snapshots to it with Terraform (as shown in Figure 2-1) to be used as a prepatch or predeployment rollback image.

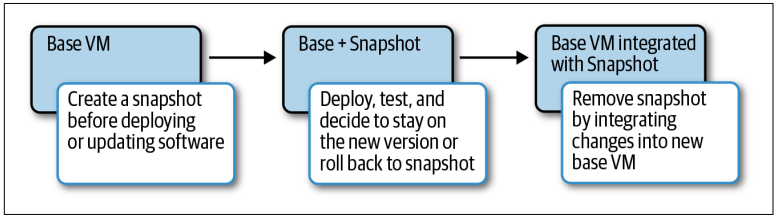

Figure 2-1. Workflow of using a snapshot

vSphere snapshots created by Terraform will include every attached disk object, including shared storage, which would not normally be captured in a snapshot using vCenter UI. This could cause both disk usage and VM performance issues.

You can add your snapshot resource code as a separate *.tf* file to make it easy to add and remove snapshot functionality to a process. Beware of runaway snapshots, and always keep track of the age of your snapshots using vCenter or other command line tools.

Here is the code for your simple VM example to add a snapshot:

```
resource "vsphere_virtual_machine_snapshot" "snap" {
  count = 1
  virtual_machine_uuid = "${vsphere_virtual_machine.vm[count.index].uuid}"
  snapshot_name        = "snap of ${vsphere_virtual_machine.vm[count.index].name}"
  description          = "Snapped by Terraform"
  memory               = "true"
  quiesce              = "true"
  remove_children      = "false"
```

```
  consolidate          = "true"
}
```

Using the `consolidate = true` option ensures that you will consolidate the delta changes in the snapshot disk instances back to the core VM when you destroy the resource.

Removing just the snapshot is done by using the `-target` parameter so that you remove only a specific part of the live Terraform configuration:

```
terraform destroy -target=vsphere_virtual_machine_snapshot.snap[0]
```

Snapshots create a safe rollback point, making deployment of new code and changes safer, and will increase the confidence of your team to deploy more often.

Terraform Taint and Untaint

Building more complex VMs and multi-VM applications adds some more challenges that introduce the need for the `taint` and `untaint` commands. You may have a vApp or set of VMs with multiple resources that make up the shared configuration. If one of those machines becomes corrupted or fails, you may not want to recreate the entire virtual application environment. This is where tainting your resource comes in handy.

Let's use an example where there are two VMs in an application resource configuration. Terraform will assign each of the resources to an array value, as illustrated in this `terraform apply` output:

```
vsphere_virtual_machine.vm[0]: Creation complete after 18s
vsphere_virtual_machine.vm[1]: Creation complete after 19s
```

You might want to re-create just one of the machines. You could do this by using a targeted `terraform destroy`:

```
terraform destroy -target=vsphere_virtual_machine.vm[0]
```

Another preferred way to do this using built-in re-creation features is to mark the resource as tainted and then to rerun the `terraform apply` process:

```
$ terraform taint vsphere_virtual_machine.vm[0]
Resource instance vsphere_virtual_machine.vm[0]
has been marked as tainted.
```

Running the `terraform plan` will show the following:

```
Terraform will perform the following actions:

  # vsphere_virtual_machine.vm[0] is tainted, so must be replaced
-/+ resource "vsphere_virtual_machine" "vm" {
…
Plan: 1 to add, 0 to change, 1 to destroy.
```

You might also find that some resources become tainted because of host failures, storage migrations, or other operational changes to the environment. You can also identify a tainted resource as safe without re-creating it by using the `terraform untaint` command:

```
$ terraform untaint vsphere_virtual_machine.vm[0]
Resource instance vsphere_virtual_machine.vm[0]
has been successfully untainted.
```

This applies to any resource in your Terraform configuration.

What We Learned

This chapter walked through a number of practical examples and code samples as a way to familiarize you with how Terraform configurations are written and how the life cycle flow works. These examples configured and dynamically updated your resources and worked with tagging, resource pools, and snapshots with little to no code required. This shows the power of Terraform as a way to operate IaC. Now that you have deployed a VM, you are ready to expand into more ways to take advantage of Terraform for your VMware environment with simple host configuration.

CHAPTER 3

Managing vSphere Host and Cluster Resources

In this chapter, we explore managing host resources with Terraform. You learn how to create and manage storage, networking, and cluster resources for your vSphere host infrastructure, which will help increase the amount of automation and consistency.

This will be useful in case of knowledge transfer to new team members, recovery of an environment in the case of losing services and resources, or building alternate environments for things like disaster recovery and test labs.

Creating vSphere Clusters

There are many other handy host-oriented features when using your existing datacenter. Out of the box, Terraform is not built for deploying a full VMware environment from bare metal, but there are lots of ways to do basic configuration using other scripts and processes and then turn over the design and deployment to Terraform.

Here's an example of using simple, declarative code to build a cluster within an existing VMware datacenter:

```
resource "vsphere_compute_cluster" "cluster" {
  name             = "discodc02-cluster"
  datacenter_id         = "${data.vsphere_datacenter.dc.id}"
  drs_enabled         = true
  drs_automation_level     = "fullyAutomated"
```

```
    ha_enabled          = true
  }
```

You can see the parameters that will be familiar to you from creating clusters manually. The DRS and HA settings are defined including the automation level of DRS, which can be `manual`, `partiallyAutomated`, or `fullyAutomated`. The automation level is optional and will use `manual` as the default if you do not specifically declare your setting.

Wait, I Do a Lot of This with PowerCLI Today…

PowerCLI has lots of capabilities to handle similar functions. What makes Terraform a powerful pairing or replacement for many of the deployment scenarios is that it handles idempotence and state natively and also requires significantly less code to do the same tasks (in most cases).

Adding Hosts to a vSphere Cluster

Creating your cluster is the first step. Next you will want to populate that cluster with hosts using just a few more lines of code. You begin by adding a data source for a set of hosts:

```
data "vsphere_host" "hosts" {
  count         = "${length(var.hosts)}"
  name          = "${var.hosts[count.index]}"
  datacenter_id = "${data.vsphere_datacenter.dc.id}"
}
```

You now assign a variable, which will be a list of hosts that you want in the cluster:

```
variable "hosts" {
  default = [
    "HOSTNAME_1",
    "HOSTNAME_2"
  ]
}
```

The last step is to add a resource section for a `vsphere_compute_cluster_host_group`, as shown here:

```
resource "vsphere_compute_cluster_host_group" "cluster_host_group" {
  name               = "terraform-test-cluster-host-group"
  compute_cluster_id  = "${vsphere_compute_cluster.compute_cluster.id}"
  host_system_ids     = "${data.vsphere_host.hosts.*.id}"
}
```

Running the Terraform configuration will take the hosts as listed in the variable and move them into the vSphere cluster (as illustrated in Figure 3-1) from the vCenter tasks console.

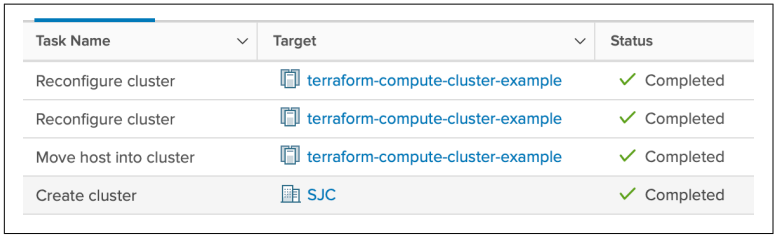

Figure 3-1. Terraform configuration

If you run a `terraform destroy` command, you will see it behave differently than it did in previous examples with VM operations. The `destroy` command on this cluster will throw an error:

```
Error: cluster "/datacenter/host/terraform-compute-cluster
-example" still has hosts or virtual machines. Please move or
remove all items before deleting
```

This is by design because of vSphere requirements and limitations. You will need to put the hosts into maintenance mode and move them out of the cluster in order to destroy it. This is a good example of certain processes that will require further intervention when doing build and tear down.

Using simple declarative code now allows you to move, change, configure, and remove host and cluster resources which is helpful for lab builds, disaster recovery environment configuration, documenting your production clusters, and much more.

CHAPTER 4

Next Steps in Your Terraform for VMware vSphere Journey

Using version control to store and manage your code is the next step in unlocking more collaborative development of IaC for you and your team. Version control is valuable even when there might be only one operator because it centralizes where the "source of truth" is rather than trusting in code stored only on local workstations or servers.

Transitioning from Local Files to Version Control

Using version control is both safer and more versatile than local file storage. Version control protects you in the case of loss of files or changes in configuration that produce undesired results. How many times have you had files named *script_1*, *script_1_working*, *script_1_working_2*, and other such names? Version control is valuable both locally and remotely because you can build your scripts on a workstation storing iterations in local version control and then push to a remote repository for off-workstation storage.

Figure 4-1 depicts a simple workflow using Git that might look a lot like how you were using file versions in the past.

This simple workflow now replaces the need for exotic filenames and having to remember which ones are working, and also pushes

the code to a central repository (aka repo) so that you can clone that code down to other workstations or servers.

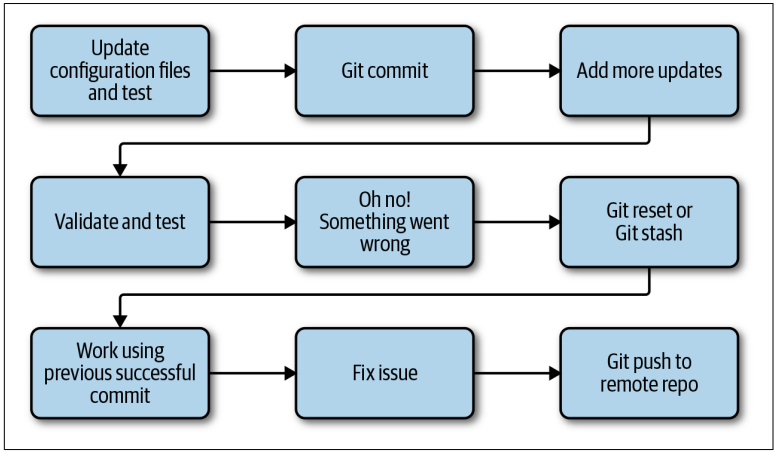

Figure 4-1. Git workflow including safely reverting in cases of error

Centralizing and Collaborating

You've seen a few ways to use simple code to replace manual processes. The culture of collaboration can really move the different parts of your IT organization to do better things, faster and safely, by embracing common tools and ensuring consistent outcomes.

Using private online repositories or on-premises code storage (e.g., GitLab) will increase the collaborative capabilities and also comes with more process workflows like building docs in Markdown and using issues to track bugs or feature requests. It's also important to remember the key risks that you will encounter along the journey.

Beware Secret Storage in Code!

Possibly the highest risk is potential public exposure. The secondary risk is the departure of team members who have local copies of code, including those carefully guarded (and not often rotated) secrets.

As you adopt more IaC practices, you can also use the same tools to help centralize and repeatedly update credentials and other secrets. The more you adapt to immutable and replaceable infrastructure, the more opportunity you have to embrace better security practices.

About the Author

Eric Wright is a technology evangelist at Turbonomic, blogs at *DiscoPosse.com*, and podcasts at *DiscoPossePodcast.com*. With a long history in the industry as a systems architect and technologist, Eric is also deeply involved in technology communities, including Microsoft, VMware, OpenStack, Kubernetes, DevOps, and many others. Eric is also the cofounder of Virtual Design Master (*VirtualDesignMaster.io*) and RapidMatter (*RapidMatter.io*), both of which are founded on the power of people and community in technology.