**17. Infrastructure Automation**

Shivakumar R Goniwada[**1**](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_17_Chapter.xhtml#Aff2)

(1)

Bangalore, India

Cloud native modern-day businesses depend on well-defined and automated IT infrastructure. Virtualization, the cloud, containers, server automation, and software-defined networking are meant to simplify operations. When organizations use the manual provisioning of infrastructure, it leads to delays in setting up an infrastructure, networking, storage, etc.

Many organizations struggle to manage manual IT tasks and processes across siloed teams. Sometimes the infrastructure team makes you wait a week or two to provide the infrastructure. There is a growing need to automate IT tasks and processes, and automation helps you to streamline them.

Automation is essential for both IT optimizations and cloud native transformations. To support business success, IT environments must be efficient, scalable, and reliable. Infrastructure operation helps you to streamline operations, improve agility, and increase security and availability.

This chapter helps you to understand how you effectively use principles, patterns, and practices through the DevOps pipeline to automate infrastructure.

In this chapter, I will cover the following:

* Infrastructure-as-code principles and patterns
* Tools and services
* Testing infrastructure changes
* What can’t you automate?

**What Is Infrastructure Automation?**

Automation is at the core of many organizations’ technology landscapes, propelled by the need to innovate faster, manage increasingly complex IT environments, accommodate new development approaches, and meet financial objectives.

Infrastructure automation is a set of processes that you use to reduce manual efforts associated with managing and provisioning workloads in the public, private, or hybrid cloud. It uses software/scripts to create repeatable instructions and processes to replace or reduce human interaction with IT systems. The tools work within the limits of instructions to perform tasks with little to no human intervention.

Automation plays a pivotal role in leveraging deployment scripts, engaging teams, monitoring tools, and tracking performance. It can also provide a better degree of reliability and boost cross-team collaboration.

By using automation, you can automate most IT tasks including the following:

* Managing physical infrastructure
* Deploying applications
* Administering virtualized environments
* Managing containers and Kubernetes environments
* Managing networks
* Implementing sanity checks and smoke tests
* Managing user access to infrastructure resources
* Troubleshooting and debugging system health
* Managing the inventory of your infrastructure resources

Automation simplifies IT infrastructure management and application service delivery by streamlining error-prone, time-consuming, and manual IT tasks and processes. Infrastructure as code (IaC) automates the provisioning of infrastructure, enabling your organization to develop, deploy, and scale cloud native services with speed, have fewer errors, and reduce costs by using various tools.

**What Can You Automate?**

You can automate most features of your infrastructure. The key to automation in the infrastructure is not just about provisioning infrastructure but also about connecting teams, processes, and tools into a single automated flow. You can automate the following along the DevOps pipeline :

* *Databases*: Hardware and servers or managed services from cloud vendors.
* *Cloud native services*: Virtualized infrastructure, containers, Kubernetes, OS, networking, and storage.
* *Development environment*: Cloud resources in similar environments as the cloud native services. The environment consists of the entire CI and CD stacks like source code management, build tools, code review tools, security scanning, Artifactory, application lifecycle management (ALM) tools, infrastructure-as-code tools, etc.
* *Test environments*: Cloud resources in similar environments as cloud native services plus testing tools, test data management, etc.

**What Is Infrastructure as Code?**

Infrastructure as code (IaC) is the engineering, managing, and provisioning of infrastructure resources through code instead of using a manual or semi-automated process to configure the system.

Provisioning infrastructure is a time-consuming and costly process and requires physically setting up hardware, installing the OS, configuring the network, etc. The virtualization, container, and cloud native environments eliminate the problem of physical servers.

IaC uses a descriptive coding language to automate the provisioning of the IT infrastructure. Virtualization, the cloud, containers, Kubernetes, servers, storage, and networks should simplify the IT operational work. It should take less time to provision, configure, update, and maintain services. Problems should be quickly identified and rectified, and the system should all be configured.

IaC is an approach to infrastructure automation based on practices from software engineering. It emphasizes consistent, repeatable routines for provisioning and changing the system and its configuration.

The changes are the biggest risk to a production system. Continuous change is inevitable due to business disruption and technology, and change is the only way to improve your system behavior. Therefore, you need to make changes accurately, reliably, and rapidly. The changes can be compliance, new features added, technical glitches, configuration changes, etc.

The following are practices you must adopt when you are implementing IaC:

* Define everything as code.
* Continuously test and deliver all work.
* Build small and incrementally.

These are the benefits of IaC:

* More quickly adapt to changes in the market
* Reduces the effort and risk of making changes to infrastructure
* Improved consistency across all environments including nonproduction environments
* Lower costs and improved ROI
* Enables engineers of infrastructure to get resources as requested and on time
* Streamlined process across teams in an organization
* Makes governance, security, and compliances visible
* Manages efficiently on load on services with spikes
* Improves the speed to troubleshoot and resolves failures and conflicts

**IaC in Build Pipeline Automation**

Organizations can deliver cloud native services through continuous delivery (CD). Businesses that embrace infrastructure, applications, and compliance outperform their peers with faster delivery, they manage risk better, and they are more assured of software security and stability. Figure [17-1](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_17_Chapter.xhtml#Fig1) shows the steps you are required to follow for infrastructure automation along the CD pipeline.



***Figure 17-1***

IaC steps

**Capture Requirements**

Each service has its features that determine where it should be deployed. Some services require higher-performance infrastructure, some services require higher CPUs, some services require a lot memory to process, and some services require high availability. Identify the key requirements of your services, and map each service against the infrastructure requirements. Depending on your IT adoption, you may choose to deploy it in the public cloud, private cloud, or hybrid cloud environment.

**Prepare Automation Code**

The requirements and mapping of these to the infrastructure provide a clear view of services, the infrastructure required, and the cloud services adopted. The next step is to identify IaC tools and create a template for every activity of infrastructure including containers, Kubernetes, OS, storage, networking, etc. The infrastructure is the underlying foundation for all IT operations. Automating the underlying infrastructure lifecycle management streamlines and improves accuracy and speed.

**Set Up Infrastructure**

The templates you create allow you to consistently deploy services across cloud environments including hybrid, private, public, etc.

In cloud services, you are required to provision VMs according to the templates; create containerization and Kubernetes workloads; and set up credentials, roles, and virtual private cloud (VPC) access for all the resources.

Provision VMs, assign an IP address, attach storage, load balance workloads, manage hosts within clusters, create housekeeping activities, and create replicas for database HA.

**Install OS**

Based on your system requirements, you are required to automate a standardized operating environment to improve efficiency and reduce costs. In the OS, use the template to automate OS images, secure settings including authentication, and manage compliances.

**Set Up Network and Storage**

Networks connect all of your services, and they must be managed to allow the right access and right bandwidth for clusters. The automation helps you make predefined, pretested changes on demand. You can use templates to automate firewall ports, access control lists (ACLs), virtual local area networks (VLANs), patches, switches, etc.

Cloud native services are based on polyglot persistence. Persistence systems like database and caching must be configured and managed to hook the right persistence to the right services. Automation in storage helps you to reduce human involvement and error. You can use the template to automate the persistence layer to services, housekeeping activity like backup and restore, etc.

**Deploy Services**

Service deployment is the last step of the IaC. As a key business asset, the services and workloads in a cloud must be configured properly to ensure optimal performance and security. Automation helps you to consistently deploy across all environments including nonproduction environments. You can use templates to automate services to install, configure and patch, load, and migrate data to the services; configure credentials; dynamically scale service resources; conduct sanity and smoke tests on deployment; and manage the lifecycle.

**Define Everything As Code**

There are many more ways to provide an infrastructure than writing code and using an IaC tool. All the cloud vendors have a nice user interface (UI); you can provision and deploy all the required services for your project. These are good for small projects, but what about the enterprise that has thousands of services?

Implementing and managing your system services as code enables you to leverage speed to improve quality. More important is that you can automate everything along with the DevSecOps pipeline.

Every infrastructure automation tool such as Terraform, Ansible, SaltStack, Chef, Puppet, etc., has a different name for source code such as *playbooks*, *cookbooks*, *manifests*, and *templates*. The infrastructure code specifies both the infrastructure elements you want and how you want them configured. You run an IaC tool to apply your code to an instance of your infrastructure. The tools either create the required new infrastructure or modify the existing one.

The following are the infrastructure elements you should define in your IaC code. These are just a few items; you should create code based on your needs.

* An infrastructure stack, either in the cloud or noncloud
* Elements of server configuration such as files, user accounts, CPU, memory, etc.
* Server role and access permissions
* A server image definition generates an image for building multiple server instances
* An application package that defines how to build a service
* Configuration of operational services such as monitoring, logging, etc.
* Validation rules such as smoke test and sanity check

**How Do You Select an IaC Tool?**

Always externalize the configuration to build the element. Usually the configuration is defined in the text-based file separately from the tools. Noncode infrastructure tools store infrastructure definitions of data that you can’t directly access. Instead, you can edit using APIs, the GUI, etc. The issue is that the noncode tools are a black box and are a drawback of versioning the code (you can do it only if the tool supports it), CI configurations for job triggers, etc.

A tool with an external code specification doesn’t constrain versioning to use specific workflows. You are free to use your project source control management tool and create a job for the same CI/CD orchestration tool as Jenkin.

**What Coding Language Can You Use?**

Earlier you might have used a scripting language like Bash, Perl Power Shell, or Ruby to automate infrastructure management tasks. CFEngine, Chef, Puppet, Ansible, and SaltStack use declarative, domain-specific languages (DSLs) for infrastructure management. Terraform and CloudFormation use the declarative DSL model. The advantages of declarative languages are that they simplify the infrastructure code by separating the infrastructure and how to implement it.

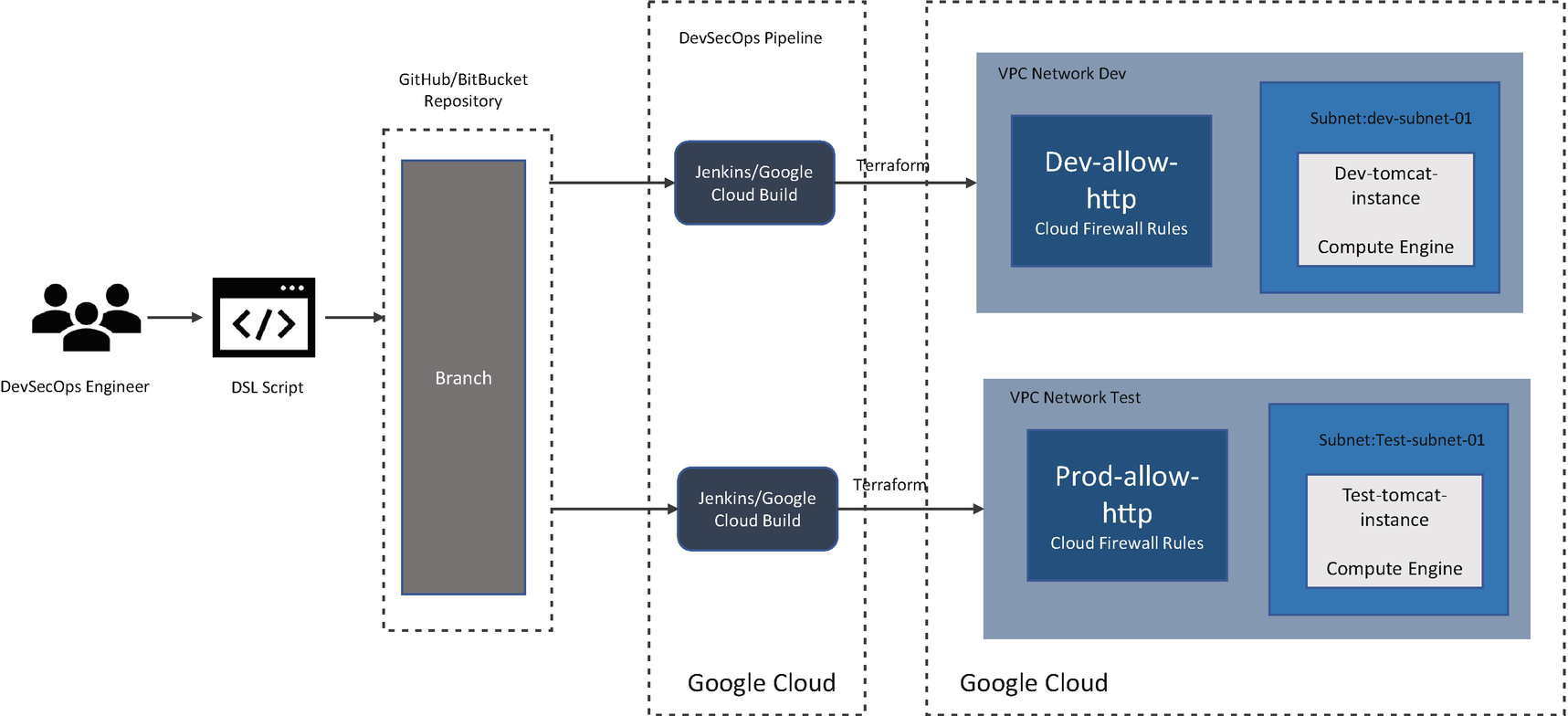
Many IaC tools use DSL. Your code defines your desired state for your infrastructure, such as which packages and user account should be on the server or how much RAM or storage you should have.

**IaC Example**

This example provides an overview of how to manage IaC with Terraform and Cloud Build/Jenkins in the Google Cloud by using GitOps.

* **GitOps***: First coined by Weaveworks, its key concept is using a Git or Bitbucket repository to store the environment state.*

Terraform is an IaC tool that uses the code to manage the infrastructure, as shown in Figure [17-2](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_17_Chapter.xhtml#Fig2). In this architecture, I am using GitOps practices for managing Terraform execution.



***Figure 17-2***

IaC architecture with Terraform and Google Cloud

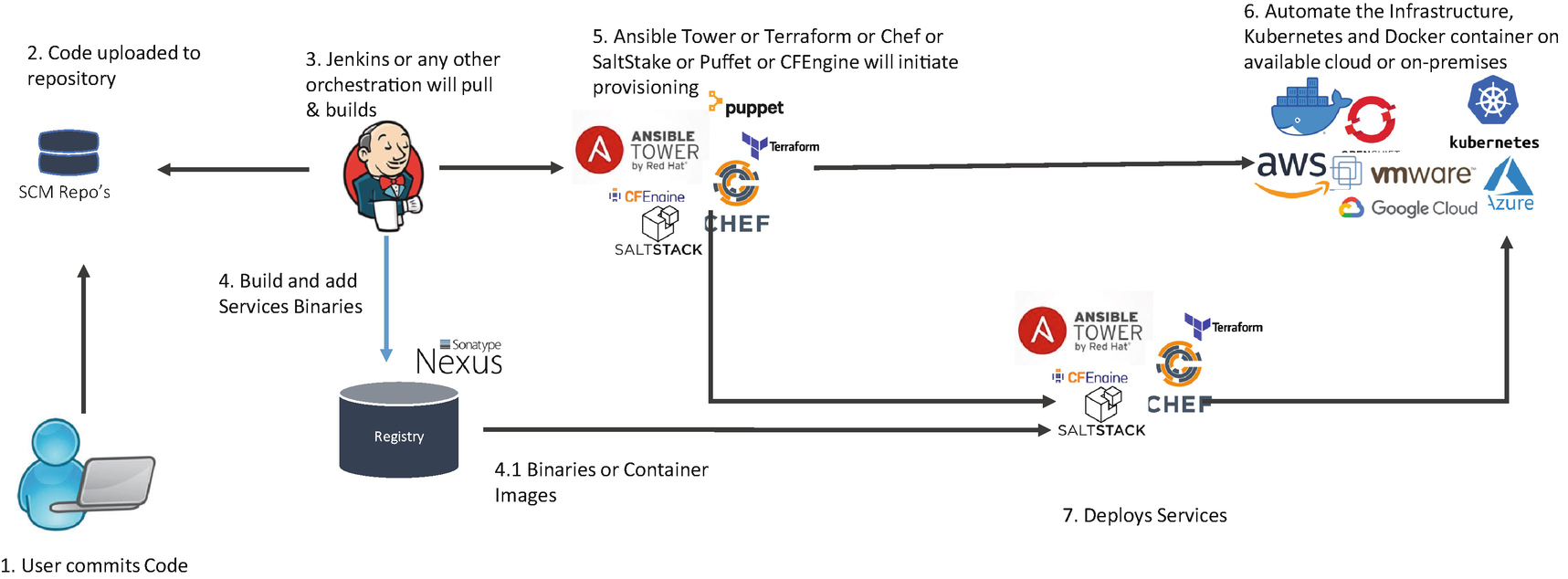
The DevOps engineer creates a Terraform script and pushes it into the Git repository with separate dev and test branches. In Figure [17-2](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_17_Chapter.xhtml#Fig2), Jenkins/CloudBuild triggers and applies Terraform manifests to achieve the state you want in the respective environment. You can use Terraform templates to create IaC scripts. This helps you to use them across environments.

**IaC Tools**

The following sections cover the IaC tools available in the industry; you can choose which is best for you. Figure [17-3](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_17_Chapter.xhtml#Fig3) illustrates how IaC tools work in your DevOps pipeline.

**Note**

Tools like Jenkins and Nexus are just two examples; you can use any other orchestration of the Artefactory tools.



***Figure 17-3***

How IaC tools integrate with the DevOps pipeline and provision infrastructure

**Terraform**

Terraform is an open source IaC tool, created by HashiCorp. It is a declarative coding tool and enables developers to use high-level HashiCorp Configuration Language (HCL) to describe the state of any cloud platform or hybrid platform. Terraform can provision infrastructure across multicloud and on-premises data centers, and it works safely and securely based on changes in a configuration like reprovisioning, adding new services to existing platforms, etc. The Terraform architecture is based on small modules; each module is a reusable Terraform configuration for multiple infrastructure resources. You can make small Terraform files that are modules, and each module can be reused and can call other modules.

The main concepts of Terraform are as follows:

* *Configurations*: Terraform uses text files to describe the infrastructure and its variables. This is called a Terraform configuration file has a .tf extension. The configuration comes in two formats: Terraform format and JSON.
* *Resources*: Resources are basic building blocks of a Terraform configuration, and resources are cloud provider specific.
* *Variables*: To make configurations more portable and flexible, Terraform supports the use of variables. By changing the variables, you can potentially reuse a single configuration file multiple times.

The following are Terraform features:

* Terraform is platform-agnostic and can work with any cloud providers or private data centers.
* Terraform creates an immutable infrastructure.
* Terraform has a planning step where it generates an execution plan. The execution plan shows what Terraform will do when you apply.
* Terraform constructs a graph for all your cloud native resources, and parallelization creation of your infrastructure and modification of any nondependent resources; therefore, it provisions the infrastructure as efficiently as possible, and operators get insight into the dependencies in their infrastructure.
* Complex changes sets can be applied to your infrastructure with no or minimal interaction. With the execution plan and resource graph, you will know exactly what Terraform will change and in what order, avoiding many errors.

The following are the main benefits of Terraform:

* Dynamic infrastructure and safe disposal of any configuration changes
* Using version control and applying testing to the infrastructure
* Standardization of the infrastructure, since no or less human intervention is required and a consistent infrastructure is available across your environment even across multicloud environments
* Validate infrastructure before deployment
* Easy to destroy a server and redeploy if your cloud native services crashed instead of repairing themselves

**Ansible**

Ansible is an open source provisioning tool for managing the configuration and application deployment tools enabling IaC. It provides an enterprise framework for building and operating an automation framework. You can centralize and control your infrastructure with a visual dashboard and role-based access control. The Ansible platform uses the YAML language for infrastructure code. Ansible’s simple, easy-to-read automation language has made it easy for teams across the organization to understand. Ansible works by connecting to your nodes, pushing out Ansible modules with programs, and executing over SSH. The commercial version of Ansible is Ansible Tower from Red Hat. The following are the benefits of Ansible:

* It’s simple to set up and use; no special coding skills are necessary to use Ansible’s playbook.
* Ansible lets you model highly complex IT workflows.
* You can orchestrate the entire application environment no matter where it is deployed.
* You don’t need to install any software agents.

**SaltStack**

SaltStack is open source provisioning, configuration management, and application deployment tool enabling IaC. It is built on Python. It uses simple human-readable YAML combined with event-driven automation to deploy and configure complex IT systems. The way it gets information about infrastructure is to query it in real time rather than rely on stale date. It is based on a master and slave architecture; the master is a lightweight set of instructions that send commands to slaves or minions with properties and asks to run commands with these arguments. The minions store properties locally and act on their own. It is designed for high performance and scalability, and the communication between master and minions is a persistent data pipe using ZeroMQ or raw TCP. The messages are asynchronously serialized on the wire using MessagePack and internally use Python Tornado as an asynchronous networking library. The following are the benefits of SaltStack:

* It’s fault tolerant. Salt minions can connect to multiple masters at one time with a YAML configuration.
* It is designed to handle 10,000 minions per master.
* Salt is easy to set up and provides single remote execution architecture.
* It is language agnostic; it can support any language.
* It is a fast, lightweight communication method to provide the foundation for a remote execution engine.

**Chef**

Chef is a configuration management tool written in the Ruby DSL language and Erlang. It uses Ruby encoding to develop basic building blocks such as recipes and cookbooks. It integrates with any of the cloud technologies. The key building blocks of Chef are recipes and cookbooks. A recipe is a collection of attributes used to manage the infrastructure. These attributes have been used to change the existing state of infrastructure. A cookbook is a collection of recipes. When Chef runs, it ensures that the recipes present inside it get a given infrastructure to the desired state. Chef works on a three-tier client-server model wherein the cookbooks are servers, the recipes are clients, and the knives are communicated across Chef. The following are the features of Chef:

* It does not work on assumptions about the current status of a node. It uses its mechanism to get the status of the machine.
* Using the Knife utility in Chef, it can integrate with any cloud infrastructure.

**Puppet**

Puppet is a configuration management tool and developed by using Ruby. This tool is written in the Ruby DSL language that helps in converting a complete infrastructure into code format. It follows the client-server model, wherein one machine in a node acts as a server, called the Puppet *master*, and the other acts as a client called the *slave* on nodes. It manages any system from scratch, from the initial configuration to the end of the lifecycle. The features of Puppet are as follows:

* It supports idempotency, which makes it unique. You can safely run the same set of configurations multiple times on the same machine.
* It works very well cross-platform with the help of a resource abstraction layer (RAL) that uses Puppet resources.
* It provides details with graphical reporting. With this you can visualize the infrastructure and communicate and quickly respond to modifications.

**CFEngine**

CFEngine is an open source configuration management system with self-healing capabilities and a desired state, with a model-oriented approach. It is suitable for managing a system composed of everything from a single host to hundreds of thousands hosts. It is based on a decentralized knowledge-based architecture. Its purpose is to implement a knowledge-based infrastructure through configuration management, and it simplifies the tasks of system configuration and maintenance. The CFEngine host acts as the policy hub, which is a server where the clients fetch their policy files. It ensures that the behavior of these clients is consistent. The following are the features of CFEngine:

* You do not need to tell it what to do. Instead, you specify the state of the system, and it automatically decides the action to take to reach the desired state.
* It defines the configuration of an entire IT system, including devices, users, applications, and services.
* You can check the system state at any given moment.
* You can ensure compliance with the desired state.
* You can propagate real-time modifications or updates across the system.

**AWS Cloud Formation**

AWS Cloud Formation gives you an easy way to model a collection of related AWS resources and other cloud resources. You can create the code from scratch by using a cloud formation template language, either in YAML or JSON format. A template describes your desired resources and their dependencies so you can launch and configure them together as a stack. You can use these templates to manage the entire stack in a single unit. You can manage the template code locally in the source code repository or upload it into the S3 bucket. Use CloudFormation via the browser console, command-line tools, or APIs to create a stack based on your template code. The following are the features of CloudFormation:

* It supports DevOps and GitOps best practices.
* You can scale your infrastructure globally. Manage resource scaling by using templates across your organization and across AWS accounts and regions.
* You can integrate with other AWS services. You can integrate other services such as AWS identity and access management for access control, AWS config for compliance, etc.
* You can manage third-party and private resources. It can manage third-party and private resources alongside your AWS services.

**IaC Tools Comparison**

Table [17-1](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_17_Chapter.xhtml#Tab1) provides you with a key comparison across major IaC tools.

***Table 17-1***

IaC Tools Comparison

| Features | Terraform | Ansible | SaltStack | Chef | Puppet | CFEngine | AWS Cloud Formation |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Tool type** | Provisioning | Config management | Config management | Config management | Config management | Provisioning | Provisioning |
| **Architecture** | Push | Push | Push and Pull | Pull | Pull | Pull | Push |
| **Provisioning approach** | Declarative | Declarative | Declarative | Declarative | Declarative | Declarative | Declarative |
| **Languages** | HashiCorp configuration language | YAML | YAML | Ruby | DSL and ERB | DSL | TS, JS, Python, Java |
| **Lifecycle (state) management** | Yes | No | No | No | No | No | No |
| **Agents** | No | No | Yes | Yes | Yes | Yes | Yes |
| **Community** | Huge | Huge | Large | Large | Large | Small | Small |
| **Cloud support** | All cloud | All cloud | All cloud | All cloud | All cloud | All cloud | AWS |

**Summary**

To get the value of cloud and infrastructure automation, you need a cloud and cloud native mindset. Automating your infrastructure takes time, especially when you are on the learning path. But doing consistently helps you to make changes.

In this chapter, I covered automation in an infrastructure by using infrastructure as code and also provided an end-to-end automation pipeline for infrastructure. I provided the IaC tools and methodology to adopt automation in your project. Finally, I provided one reference implementation of how we used IaC in our project.