1
DevOps Culture and Practices

DevOps, a term that we hear more and more in enterprises with phrases such as *We do DevOps* or *We use DevOps tools*, is the contraction of the words Development and Operations.

DevOps is a culture different from traditional corporate cultures and requires a change in mindset, processes, and tools. It is often associated with **continuous integration** (**CI**) and **continuous delivery** (**CD**) practices, which are software engineering practices, but also with **Infrastructure as Code** (**IaC**), which consists of *codifying* the structure and configuration of infrastructure.

In this chapter, we will see what DevOps culture is, what DevOps principles are, and the benefits it brings to a company. Then, we will explain CI/CD practices and, finally, we will detail IaC with its patterns and practices.

In this chapter, the following topics will be covered:

- Getting started with DevOps
- Implementing CI/CD and continuous deployment
- Understanding IaC

Getting started with DevOps

The term DevOps was introduced in 2007-2009 by Patrick Debois, Gene Kim, and John Willis, and it represents the combination of **Development** (**Dev**) and **Operations** (**Ops**). It has given rise to a movement that advocates bringing developers and operations together within teams. This is to be able to deliver added business value to users more quickly and hence be more competitive in the market.

DevOps culture is a set of practices that reduce the barriers between developers, who want to innovate and deliver faster, on the one side and, on the other side, operations, who want to guarantee the stability of production systems and the quality of the system changes they make.

DevOps culture is also the extension of agile processes (scrum, XP, and so on), which make it possible to reduce delivery times and already involve developers and business teams, but are often hindered because of the non-inclusion of Ops in the same teams.

The communication and this link between Dev and Ops does, therefore, allow a better follow-up of end-to-end production deployments and more frequent deployments of a better quality, saving money for the company.

To facilitate this collaboration and improve communication between Dev and Ops, there are several key elements in the processes to be put in place, as in the following examples:

- More frequent application deployments with integration and continuous delivery (called **CI/CD**)
- The implementation and automation of unitary and integration tests, with a process focused on **Behavior-Driven Design** (**BDD**) or **Test-Driven Design** (**TDD**)
- The implementation of a means of collecting feedback from users
- Monitoring applications and infrastructure

The DevOps movement is based on three axes:

- **The culture of collaboration**: This is the very essence of DevOps—the fact that teams are no longer separated by silos specialization (one team of developers, one team of Ops, one team of testers, and so on), but, on the contrary, these people are brought together by making multidisciplinary teams that have the same objective: to deliver added value to the product as quickly as possible.
- **Processes**: To expect rapid deployment, these teams must follow development processes from agile methodologies with iterative phases that allow for better functionality quality and rapid feedback. These processes should not only be integrated into the development workflow with continuous integration but also into the deployment workflow with continuous delivery and deployment. The DevOps process is divided into several phases:
    - The planning and prioritization of functionalities
    - Development
    - Continuous integration and delivery
    - Continuous deployment
    - Continuous monitoring

  These phases are carried out cyclically and iteratively throughout the life of the project.

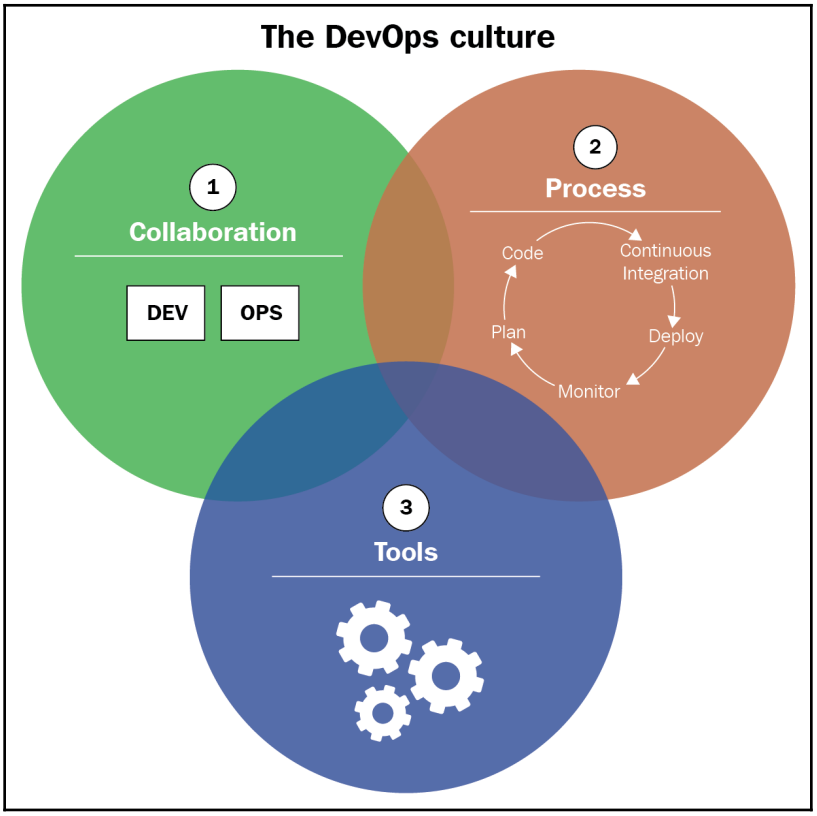
- **Tools**: The choice of tools and products used by teams is very important in DevOps. Indeed, when teams were separated into Dev and Ops, each team used their specific tools—deployment tools for developers and infrastructure tools for Ops—which further widened communication gaps.

With teams that bring development and operations together, and with this culture of unity, the tools used must be usable and exploitable by all members.

Developers need to integrate with monitoring tools used by Ops teams to detect performance problems as early as possible and with security tools provided by Ops to protect access to various resources.

Ops, on the other hand, must automate the creation and updating of the infrastructure and integrate the code into a code manager; this is called **Infrastructure as Code**, but this can only be done in collaboration with developers who know the infrastructure needed for applications. Ops must also be integrated into application release processes and tools.

The following diagram illustrates the three axes of DevOps culture—the collaboration between Dev and Ops, the processes, and the use of tools:



So, we can go back to DevOps culture with Donovan Brown's definition (`http://donovanbrown.com/post/what-is-devops`):

> *"DevOps is the union of people, process, and products to enable continuous delivery of value to our end users."*

The benefits of establishing a DevOps culture within an enterprise are as follows:

- Better collaboration and communication in teams, which has a human and social impact within the company.
- Shorter lead times to production, resulting in better performance and end user satisfaction.

- Reduced infrastructure costs with IaC.
- Significant time saved with iterative cycles that reduce application errors and automation tools that reduce manual tasks, so teams focus more on developing new functionalities with added business value.

For more information about DevOps culture and its impact on and transformation of enterprises, read the book by Gene Kim and Kevin Behr, *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win*, and *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations* by Gene Kim, Jez Humble, Patrick Debois, and John Willis.

Implementing CI/CD and continuous deployment

We saw earlier that one of the key DevOps practices is the process of integration and continuous delivery, also called CI/CD. In fact, behind the acronyms of CI/CD, there are three practices:

- **Continuous integration** (**CI**)
- **Continuous delivery** (**CD**)
- **Continuous deployment**

What does each of these practices correspond to? What are their prerequisites and best practices? Are they applicable to all?

Let's look in detail at each of these practices, starting with continuous integration.

Continuous integration (CI)

In the following definition given by Martin Fowler, there are three key things mentioned, *members of a team*, *integrate*, and *as quickly as possible:*

> *"Continuous Integration is a software development practice where members of a team integrate their work frequently... Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible."*

That is, CI is an automatic process that allows you to check the completeness of an application's code every time a team member makes a change. This verification must be done as quickly as possible.

We see DevOps culture in CI very clearly, with the spirit of collaboration and communication, because the execution of CI impacts all members in terms of work methodology and therefore collaboration; moreover, CI requires the implementation of processes (branch, commit, pull request, code review, and so on) with automation that is done with tools adapted to the whole team (Git, Jenkins, Azure DevOps, and so on). And finally, CI must run quickly to collect feedback on code integration as soon as possible and hence be able to deliver new features more quickly to users.

Implementing CI

To set up CI, it is, therefore, necessary to have a **Source Code Manager** (**SCM**) that will allow the centralization of the code of all members. This code manager can be of any type: Git, SVN, or **Team Foundation Source Control** (**TFVC**). It's also important to have an automatic build manager (CI server) that supports continuous integration such as Jenkins, GitLab CI, TeamCity, Azure Pipelines, GitHub Actions, Travis CI, Circle CI, and so on.

In this book, we will use Git as an SCM, and we will look a little more deeply into its concrete uses.

Each team member will work on the application code daily, iteratively and incrementally (such as in agile and scrum methods). Each task or feature must be partitioned from other developments with the use of branches.

Regularly, even several times a day, members archive or commit their code and preferably with small commits (trunks) that can easily be fixed in the event of an error. This will, therefore, be integrated into the rest of the code of the application with all of the other commits of the other members.

This integration of all the commits is the starting point of the CI process.

This process, executed by the CI server, must be automated and triggered at each commit. The server will retrieve the code and then do the following:

- Build the application package—compilation, file transformation, and so on.
- Perform unit tests (with code coverage).

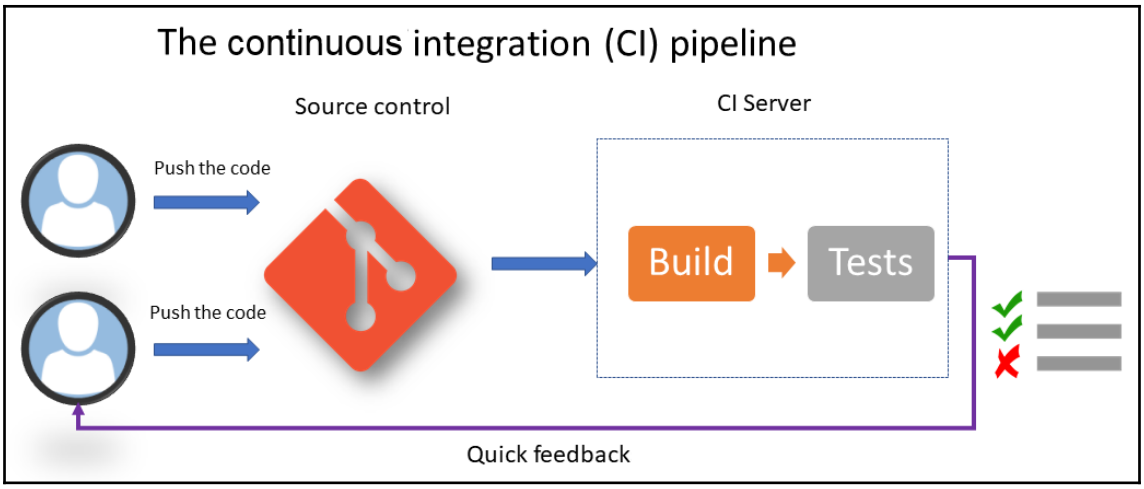
It is also possible to enrich the process with static code and vulnerability analysis, which we will look at in `Chapter 10`, *Static Code Analysis with SonarQube*, which is dedicated to testing.

This CI process must be optimized as soon as possible so that it can run fast and developers can have quick feedback on the integration of their code. For example, code that is archived and does not compile or whose test execution fails can impact and block the entire team.

Sometimes, bad practices can result in the failure of tests in the CI, deactivating the test execution, taking as arguments: *it is not serious*, *it is necessary to deliver quickly*, or *the code that compiles it is essential*.

On the contrary, this practice can have serious consequences when the errors detected by the tests are revealed in production. The time saved during CI will be lost on fixing errors with hotfixes and redeploying them quickly with stress. This is the opposite of DevOps culture with poor application quality for end users and no real feedback, and, instead of developing new features, we spend time correcting errors.

With an optimized and complete CI process, the developer can quickly fix their problem and improve their code or discuss it with the rest of the team and commit their code for a new integration:



This diagram shows the cyclical steps of continuous integration that include the code being pushed into the SCM by the team members and the execution of the build and test by the CI server. And the purpose of this fast process is to provide rapid feedback to members.

We have just seen what continuous integration is, so now let's look at continuous delivery practices.

Continuous delivery (CD)

Once continuous integration has been successfully completed, the next step is to deploy the application automatically in one or more non-production environments, which is called **staging**. This process is called **continuous delivery** (**CD**).

CD often starts with an application package prepared by CI, which will be installed according to a list of automated tasks. These tasks can be of any type: unzip, stop and restart service, copy files, replace configuration, and so on. The execution of functional and acceptance tests can also be performed during the CD process.

Unlike CI, CD aims to test the entire application with all of its dependencies. This is very visible in microservices applications composed of several services and APIs; CI will only test the microservice under development while, once deployed in a staging environment, it will be possible to test and validate the entire application as well as the APIs and microservices that it is composed of.

In practice, today, it is very common to link CI with CD in an *integration* environment; that is, CI deploys at the same time in an environment. It is indeed necessary so that developers can have at each commit not only the execution of unit tests but also a verification of the application as a whole (UI and functional), with the integration of the developments of the other team members.

It is very important that the package generated during CI and that will be deployed during CD is the same one that will be installed on all environments, and this should be the case until production. However, there may be configuration file transformations that differ depending on the environment, but the application code (binaries, DLL, and JAR) must remain unchanged.

This *immutable*, unchangeable character of the code is the only guarantee that the application verified in an environment will be of the same quality as the version deployed in the previous environment and the same one that will be deployed in the next environment. If changes (improvements or bug fixes) are to be made to the code following verification in one of the environments, once done, the modification will have to go through the CI and CD cycle again.

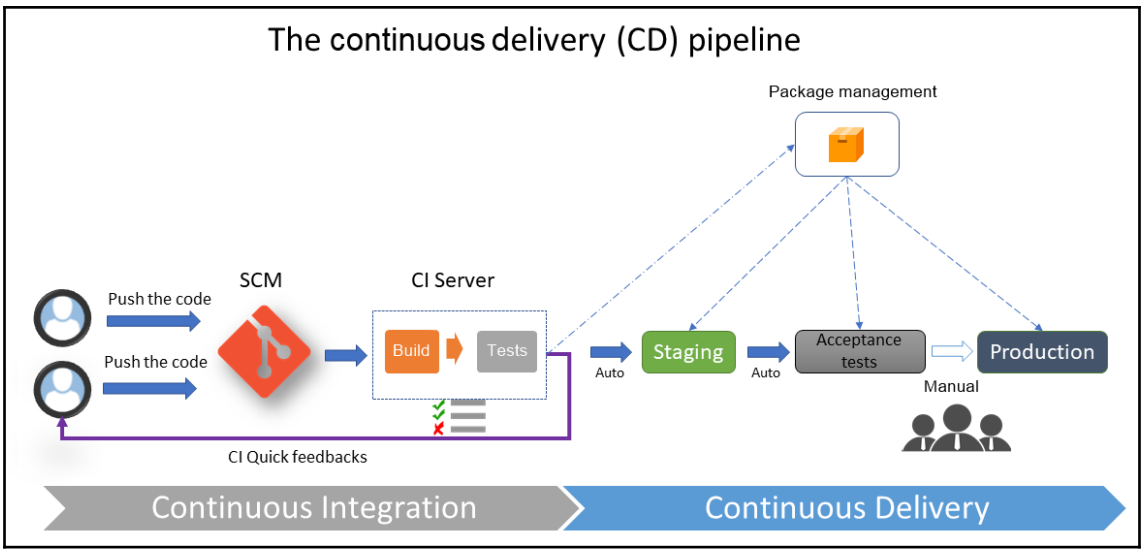
The tools set up for CI/CD are often completed with others solutions, which are as follows:

- **A package manager**: This constitutes the storage space of the packages generated by CI and recovered by CD. These managers must support feeds, versioning, and different types of packages. There are several on the market, such as Nexus, ProGet, Artifactory, and Azure Artifacts.
- **A configuration manager**: This allows you to manage configuration changes during CD; most CD tools include a configuration mechanism with a system of variables.

In CD, the deployment of the application in each staging environment is triggered as follows:

- It can be triggered automatically, following a successful execution on a previous environment. For example, we can imagine a case where the deployment in the pre-production environment is automatically triggered when the integration tests have been successfully performed in a dedicated environment.
- It can be triggered manually, for sensitive environments such as the production environment, following a manual approval by a person responsible for validating the proper functioning of the application in an environment.

What is important in a CD process is that the deployment to the production environment, that is, to the end user, is triggered manually by approved users:



This diagram clearly shows that the CD process is a continuation of the CI process. It represents the chain of CD steps, which are automatic for staging environments but manual for production deployments. It also shows that the package is generated by CI and is stored in a package manager and that it is the same package that is deployed in different environments.

Now that we've looked at CD, let's look at continuous deployment practices.

Continuous deployment

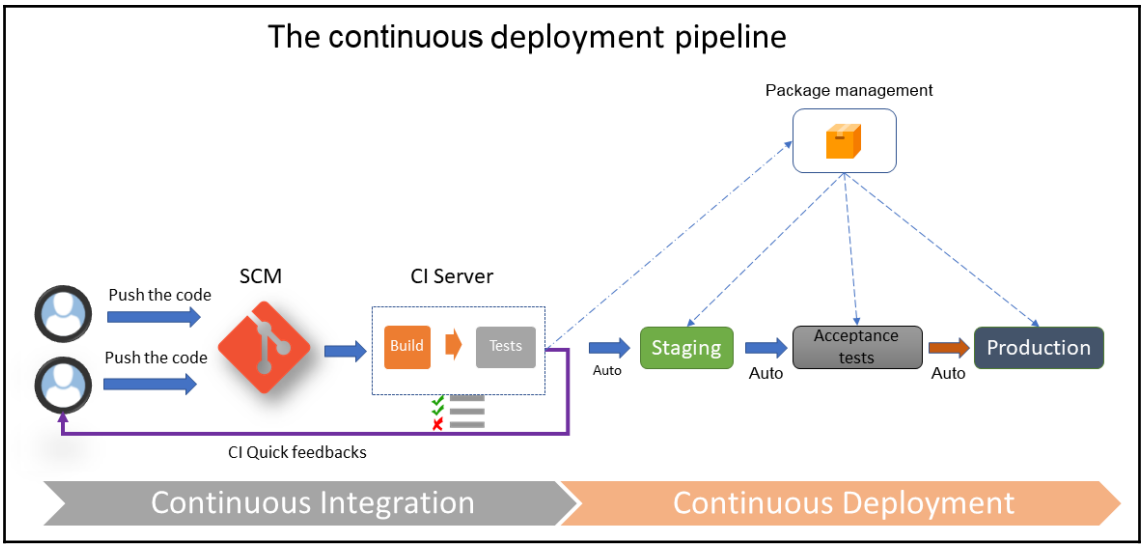
Continuous deployment is an extension of CD, but this time, with a process that automates the entire CI/CD pipeline from the moment the developer commits their code to deployment in production through all of the verification steps.

This practice is rarely implemented in enterprises because it requires a wide coverage of tests (unit, functional, integration, performance, and so on) for the application, and the successful execution of these tests is sufficient to validate the proper functioning of the application with all of these dependencies, but also automated deployment to a production environment without any approval action.

The continuous deployment process must also take into account all of the steps to restore the application in the event of a production problem.

Continuous deployment can be implemented with the use and implementation of feature toggle techniques (or feature flags), which involves encapsulating the application's functionalities in *features* and activating its *features* on demand, directly in production, without having to redeploy the code of the application.

Another technique is to use a *blue-green* production infrastructure, which consists of two production environments, one *blue* and one *green.* We first deploy to the *blue* environment, then to the *green*; this will ensure that there is no downtime required:



We will look at the feature toggle and blue-green deployment usage in more detail in `Chapter 13`, *Reducing Deployment Downtime*.

The preceding diagram is almost the same as that of CD, but with the difference that it depicts automated end-to-end deployment.

CI/CD processes are therefore an essential part of DevOps culture, with CI allowing teams to integrate and test the coherence of its code and to obtain quick feedback very regularly. CD automatically deploys on one or more staging environments and hence offers the possibility to test the entire application until it is deployed in production.

Finally, continuous deployment automates the deployment of the application from commit to the production environment.

We will see how to implement all of these processes in practice with Jenkins, Azure DevOps, and GitLab CI in `Chapter 6`, *Continuous Integration and Continuous Delivery*.

In this section, we have discussed practices essential to DevOps culture, which are continuous integration, continuous delivery, and continuous deployment.

In the next section, we will go into detail about another DevOps practice, which is IaC.

Understanding IaC practices

IaC is a practice that consists of writing the code of the resources that make up an infrastructure.

This practice began to take effect with the rise of DevOps culture and with the modernization of cloud infrastructure. Indeed, Ops teams that deploy infrastructures manually take time to deliver infrastructure changes due to inconsistent handling and the risk of errors. Also, with the modernization of the cloud and its scalability, the way an infrastructure is built requires a review of provisioning and change practices by adapting a more automated method.

IaC is the process of writing the code of the provisioning and configuration steps of infrastructure components to automate its deployment in a repeatable and consistent manner.

Before we look at the use of IaC, we will see what the benefits of this practice are.

The benefits of IaC

The benefits of IaC are as follows:

- The standardization of infrastructure configuration reduces the risk of error.
- The code that describes the infrastructure is versioned and controlled in a source code manager.
- The code is integrated into CI/CD pipelines.
- Deployments that make infrastructure changes are faster and more efficient.
- There's better management, control, and a reduction in infrastructure costs.

IaC also brings benefits to a DevOps team by allowing Ops to be more efficient on infrastructure improvement tasks rather than spending time on manual configuration and by giving Dev the possibility to upgrade their infrastructures and make changes without having to ask for more Ops resources.

IaC also allows the creation of self-service, ephemeral environments that will give developers and testers more flexibility to test new features in isolation and independently of other environments.

IaC languages and tools

The languages and tools used to code the infrastructure can be of different types; that is, scripting and declarative types.

Scripting types

These are scripts such as Bash, PowerShell, or any other languages that use the different clients (SDKs) provided by the cloud provider; for example, you can script the provisioning of an Azure infrastructure with the Azure CLI or Azure PowerShell.

For example, here is the command that creates a resource group in Azure:

- Using the Azure CLI (the documentation is at `https://bit.ly/2V1OfxJ`), we have the following:

```
az group create –location westeurope –name MyAppResourcegroup
```

- Using Azure PowerShell (the documentation is at `https://bit.ly/2VcASeh`), we have the following:

```
New-AzResourceGroup –Name MyAppResourcegroup –Location westeurope
```

The problem with these languages and tools is that they require a lot of lines of code because we need to manage the different states of the manipulated resources and it is necessary to write all of the steps of the creation or update of the desired infrastructure.

However, these languages and tools can be very useful for tasks that automate repetitive actions to be performed on a list of resources (selection and query) or that require complex processing with a certain logic to be performed on infrastructure resources such as a script that automates the deletion of VMs that carry a certain tag.

Declarative types

These are languages in which it is sufficient to write the state of the desired system or infrastructure in the form of configuration and properties. This is the case, for example, for Terraform and Vagrant from HashiCorp, Ansible, the Azure ARM template, PowerShell DSC, Puppet, and Chef. The user only has to write the final state of the desired infrastructure and the tool takes care of applying it.

For example, the following is the Terraform code that allows you to define the desired configuration of an Azure resource group:

```
resource "azurerm_resource_group" "myrg" {
     name = "MyAppResourceGroup"
     location = "West Europe"

     tags = {
         environment = "Bookdemo"
    }
}
```

In this example, if you want to add or modify a tag, just modify the `tags` property in the preceding code and Terraform will do the update itself.

Here is another example that allows you to install and restart `nginx` on a server using Ansible:

```
---
- hosts: all
 tasks:
 - name: install and check nginx latest version
 apt: name=nginx state=latest
 - name: start nginx
 service:
 name: nginx
 state: started
```

And to ensure that the service is not installed, just change the preceding code, with `service` as an absent value and the `state` property with the `stopped` value:

```
---
- hosts: all
 tasks:
 - name: stop nginx
 service:
 name: nginx
 state: stopped
 - name: check nginx is not installed
 apt: name=nginx state=absent
```

In this example, it was enough to change the `state` property to indicate the desired state of the service.

For details regarding the use of Terraform and Ansible, see `Chapter 2`, *Provisioning Cloud Infrastructure with Terraform*, and `Chapter 3`, *Using Ansible for Configuring IaaS Infrastructure*.

The IaC topology

In a cloud infrastructure, IaC is divided into several typologies:

- The deployment and provisioning of the infrastructure
- The server configuration and templating
- The containerization
- The configuration and deployment in Kubernetes

Let's deep dive into each topology.

The deployment and provisioning of the infrastructure

Provisioning is the act of instantiating the resources that make up the infrastructure. They can be of the **Platform as a Service** (**PaaS**) and serverless resource types, such as a web app, Azure function, or Event Hub but also the entire network part that is managed, such as VNet, subnets, routing tables, or Azure Firewall. For virtual machine resources, the provisioning step only creates or updates the VM cloud resource but not its content.

There are different provisioning tools such as Terraform, the ARM template, AWS Cloud training, the Azure CLI, Azure PowerShell, and also Google Cloud Deployment Manager. Of course, there are many more, but it is difficult to mention them all. In this book, we will look at, in detail, the use of Terraform to provide an infrastructure.

Server configuration

This step concerns the configuration of virtual machines, such as the configuration of hardening, directories, disk mounting, network configuration (firewall, proxy, and so on), and middleware installation.

There are different configuration tools, such as Ansible, PowerShell DSC, Chef, Puppet, and SaltStack. Of course, there are many more, but, in this book, we will look at, in detail, the use of Ansible to configure a virtual machine.

To optimize server provisioning and configuration times, it is also possible to create and use server models, also called images, that contain all of the configuration (hardening, middleware, and so on) of the servers. It will be during the provisioning of the server that we will indicate the template to use, and hence, we will have, in a few minutes, a configured server ready to be used.

There are also many IaC tools for creating server templates, such as aminator (used by Netflix) or HashiCorp Packer.

Here is an example of Packer file code that creates an Ubuntu image with package updates:

```
{
"builders": [{
     "type": "azure-arm",
     "os_type": "Linux",
     "image_publisher": "Canonical",
     "image_offer": "UbuntuServer",
     "image_sku": "16.04-LTS",
     "managed_image_resource_group_name": "demoBook",
     "managed_image_name": "SampleUbuntuImage",
     "location": "West Europe",
     "vm_size": "Standard_DS2_v2"
 }],
 "provisioners": [{
     "execute_command": "chmod +x {{ .Path }}; {{ .Vars }} sudo -E sh '{{
.Path }}'",
     "inline": [
     "apt-get update",
     "apt-get upgrade -y",
     "/usr/sbin/waagent -force -deprovision+user && export HISTSIZE=0 &&
sync"
 ],
 "inline_shebang": "/bin/sh -x",
 "type": "shell"
 }]
}
```

This script creates a template image for the `Standard_DS2_V2` virtual machine based on the Ubuntu OS (the `builders` section). Additionally, Packer will update all packages during the creation of the image with the `apt-get update` command and, after this execution, Packer deprovisions the image to delete all user information (the `provisioners` section).

The Packer part will be discussed in detail in `Chapter 4`, *Optimizing Infrastructure Deployment with Packer*.

Immutable infrastructure with containers

Containerization consists of deploying applications in containers instead of deploying them in VMs.

Today, it is very clear that the container technology to be used is Docker and that the configuration of a Docker image is also done in code in a Dockerfile. This file contains the declaration of the base image, which represents the *bone* to be used, the installation of additional middleware to be installed on the image, only the files and binaries necessary for the application, and the network configuration of the ports. Unlike VMs, containers are said to be immutable; the configuration of a container cannot be modified during its execution.

Here is a simple example of a Dockerfile:

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y nginx
ENTRYPOINT ["/usr/sbin/nginx","-g","daemon off;"]
EXPOSE 80
```

In this Docker image, we use a basic Ubuntu image, install `nginx`, and expose port `80`.

The Docker part will be discussed in detail in `Chapter 7`, *Containerizing Your Application with Docker*.

Configuration and deployment in Kubernetes

Kubernetes is a container orchestrator—it is the technology that most embodies IaC, in my opinion, because the way it deploys containers, the network architecture (load balancer, ports, and so on), and the volume management, as well as the protection of sensitive information, are described completely in the YAML specification files.

Here is a simple example of a YAML specification file:

```
apiVersion: apps/v1
kind: Deployment
metadata:
   name: nginx-demo
   labels:
       app: nginx
spec:
   replicas: 2
   selector:
     matchLabels:
        app: nginx
   template:
   metadata:
      labels:
        app: nginx
  spec:
    containers:
    - name: nginx
        image: nginx:1.7.9
      ports:
      - containerPort: 80
```

We can see in the preceding specification file, the name of the image to deploy (`ngnix`), the port to open (`80`), and the number of replicas (`2`).

The Kubernetes part will be discussed in detail in `Chapter 8`, *Managing Containers Effectively with Kubernetes*.

IaC best practices

IaC, like software development, requires the implementation of practices and processes that allow the evolution and maintenance of the infrastructure code.

Among these practices are those of software development, as in these examples:

- Have good principles of nomenclature.
- Do not overload the code with unnecessary comments.
- Use small functions.
- Implement error handling.

To learn more about good software development practices, read the excellent book, which is, for my part, a reference on the subject, *Clean Code* by Robert Martin.

But there are more specific practices that I think deserve more attention:

- **Everything must be automated in the code**: When doing IaC, it is indeed necessary to code and automate all of the provisioning steps and not to leave manual steps out of code that *distort* the automation of the infrastructure and that can generate errors. And if necessary, do not hesitate to use several tools such as Terraform and Bash with the Azure CLI scripts.
- **The code must be in a source control manager**: The infrastructure code must also be in an SCM to be versioned, tracked, merged, and restored, and hence have better visibility of the code between Dev and Ops.
- **The infrastructure code must be with the application code**: In some cases, this may be difficult, but if possible, it is much better to place the infrastructure code in the same repository as the application code. This is to have a better work organization between developers and operations, who will share the same workspace.
- **Separation of roles and directories**: It is good to separate the code from the infrastructure according to the role of the code, so you can create one directory for provisioning and for configuring VMs and another directory that will contain the code for testing the integration of the complete infrastructure.
- **Integration into a CI/CD process**: One of the goals of IaC is to be able to automate the deployment of the infrastructure, so from the beginning of its implementation, it is necessary to set up a CI/CD process that will integrate the code, test it, and deploy it in different environments. Some tools, such as Terratest, allow you to write tests on infrastructure code. One of the best practices is to integrate the CI/CD process of the infrastructure into the same pipeline as the application.
- **The code must be idempotent**: The execution of the infrastructure deployment code must be idempotent; that is, automatically executable at will. This means that scripts must take into account the state of the infrastructure when running it and not generate an error if the resource to be created already exists or if a resource to be deleted has already been deleted. We will see that declarative languages, such as Terraform, take on this aspect of idempotence natively. The code of the infrastructure, once fully automated, must allow the construction and complete destruction of the application infrastructure.

- **To be used as documentation**: The code of the infrastructure must be clear and must be able to serve as documentation. Indeed, infrastructure documentation takes a long time to be written and in many cases, it is not updated as the infrastructure evolves.
- **The code must be modular**: In an infrastructure, the components very often have the same code—the only difference is the value of their properties. Also, these components are used several times in the company's applications. It is therefore important to optimize the writing times of code, by factoring it with modules (or roles, for Ansible) that will be called as functions. Another advantage of using modules is the ability to standardize resource nomenclature and compliance on some properties.
- **Having a development environment**: The problem with IaC is that it is difficult to test its infrastructure code under development in environments used for integration and to test the application because changing the infrastructure can have an impact. It is therefore important to have a development environment even for IaC that can be impacted or even destroyed at any time.

For local infrastructure tests, some tools simulate a local environment, such as Vagrant (from HashiCorp), so you should use them to test code scripts as much as possible.

Of course, the full list of good practices is longer than this list; all methods and processes of software engineering practices are also applicable.

IaC is, therefore, like CI/CD processes, a key practice of DevOps culture that allows, by writing code, the deployment and configuration of an infrastructure. However, IaC can only be effective with the use of appropriate tools and the implementation of good practices.

Summary

In this first chapter, we saw that DevOps culture is a story of collaboration, processes, and tools. Then, we detailed the different steps of the CI/CD process and explained the difference between and that continuous deployment.

Finally, the last part explained how to use IaC, with its best practices.

In the next chapter, we will start with the implementation of IaC and how to provision an infrastructure with Terraform.

Questions

1. Of which words is DevOps a contraction?
2. Is DevOps a term that represents: the name of a tool, a culture or a society, or the title of a book?
3. What are the three axes of DevOps culture?
4. What is the objective of continuous integration?
5. What is the difference between continuous delivery and continuous deployment?
6. What is IaC?

Further reading

If you want to know more about DevOps culture, here are some resources:

- The DevOps Resource Center (Microsoft resources): `https://docs.microsoft.com/en-us/azure/devops/learn/`
- 2018 State of DevOps Report (by Puppet): `https://puppet.com/resources/whitepaper/state-of-devops-report`