CHAPTER 1

Introduction to DevOps

DevOps, DevOps, DevOps, there is hardly a day in our professional lives that we don't hear that mantra. The reason is simple: by adopting DevOps practices, a company can reduce "time to market," the time necessary to identify a new software or other requirement and to put it in place.

In this chapter, I introduce some of the advantages of DevOps and the changes that must be made to accommodate and promote their use most profitably by a company.

What Is DevOps?

The term *DevOps* is derived from the combination of two words: *developer* and *operations*. *DevOps* is used to define a movement born of the need for reducing barriers between the development and operations teams of a company. The goal of DevOps is to reduce time to market. This means adopting DevOps practices, to reduce the time needed, from the identification of a new requirement to the time it goes live for customers. The DevOps journey introduces such practices as continuous integration and continuous delivery, which help to reduce time to market and produce better quality software.

The most significant and possibly most expensive failures related to a new release occurred in 2012, when Knight Capital Group, a global financial services firm, lost $440 million, owing to a glitch in its servers during the release process, and, in 2013, when an upgrade failure at Goldman Sachs sent orders accidentally, resulting in the loss of what was thought to have been millions of dollars. DevOps allows a set of practices that can reduce potentially expensive errors.

When we think of DevOps, we must think of a movement formed to change how an entire company works together. The goal is to build a set of practices that can be used to reduce impediments to communication across different departments in the company. To be successful, DevOps should be promoted at the highest level of the company and accepted by each of its departments.

P. Riti, *Pro DevOps with Google Cloud Platform*, https://doi.org/10.1007/978-1-4842-3897-4_1

The philosophy behind DevOps was born in 2008, at the Agile conference in Toronto, Canada. During this conference, Patrick Debois presented his talk "Infrastructure and Operations." In it, Debois expounded on the application of Agile methodologies for building an infrastructure. He offered three case studies and, at the end, proposed a better method of communication and other improvements allowing developers to gain more IT skills, in the sense of the system knowledge necessary to ensure a smoother release process.

The movement grew, until the 2014 publication of the book *The Phoenix Project*. In this book, authors Gene Kim, Kevin Behr, and George Spafford describe how to use DevOps to create a successful business. With this book, the start of the DevOps movement became official, and after, a new IT figure was born with the advent of the DevOps engineer.

The DevOps Engineer

The role of the DevOps engineer is a very recent one. The DevOps engineer represents a kind of bridge between the developer and the operations manager. In most cases, the role the DevOps engineer assumes is a mix of operations manager and developer, and this is fine, because these engineers must have the necessary knowledge for advising and managing a problem from both disciplines.

In some cases, the responsibility of a DevOps engineer is connected to continuous integration and delivery. Another responsibility associated with this position is infrastructure management, usually, infrastructure as code (IaC), and to help put in place optimal DevOps practices across the company.

Some companies view the DevOps engineer as an evolution of the site reliability engineer (SRE), whose main responsibility is to maintain the software in production and automate all the steps to solve an issue when one arises and to take the appropriate steps to ensure the normal administration of tasks necessary for maintaining the system. The role of the DevOps engineer is varied and can change from one company to another, but all have at their core ensuring the changes required to adopt the DevOps practices initiated by the company.

Adopting DevOps

Adopting DevOps in a company is like starting on a new journey. During this journey, management must be effective for change to be successful. The following bulleted list highlights the essential signposts of the journey.

- The manager must promote the change.
- The developer must be responsible for the software.
- The operational people must be treated as "first-class citizens."
- Continuous integration and continuous delivery policies must be built.
- Barriers to the IT department must be removed.
- The release process must be automated.
- Agile practices must be promoted across the entire company.

To achieve the desired result requires that DevOps changes be initiated by management and integrated into the company culture.

This step is very important, to ensure a successful DevOps journey, and, of course, it involves some technical changes, to be really effective. Let's examine in detail what this means.

The Manager Must Promote the Change

To be successful, the changes required by DevOps must be pushed and accepted by management first. For change to be effective, it must have strong company approval.

Imagine, for example, at the outset of our journey, that we start a new development. We want to design a continuous integration/continuous delivery (CI/CD) system. To do that, we decide to adopt Scrum as our methodology, instead of Waterfall, which was previously used.

One day, the chief technology officer (CTO) proposes a new feature that he/she absolutely wants to release in spring, only months away, but to do this, we must cancel and delay some other features. The Scrum Master tries to tell the CTO not to introduce the new feature so soon, because it will delay other features and cause some issues. The CTO insists, and using his/her power, pushes the feature to spring. To hit this date, the team must postpone some other jobs and work faster on the new feature, thereby

creating some software quality issues. The requirements are not completely clear, and during the CI/CD, cycle issues are identified, and this makes the software of very poor quality and essentially not ready for release.

In the end, the team loses confidence in the DevOps practice, and little by little, everyone goes back to the usual way of doing things.

The Developer Must Be Responsible for the Software

In the normal development cycle, the responsibility of the development team ends when the software is released to live production. After that, the developer works on new features and is involved only when operations find a bug to fix. But this means a new feature must be released, and the operations team must find a way to mitigate the bug.

If we want to have a successful DevOps journey, we must empower the developer. This means that when the operations team finds a bug in the software, the developer working on the function must be involved in the fix. This has two major advantages:

- The developer can more easily identify the problem and find a fix for the issue. Because he/she knows the software, it is easier for him/her to find the root cause of the problem.
- Because the developer can identify a problem's root cause, it is easier for him/her to work on a permanent solution. This, with a CI/CD practice in place, reduces the time to market for release and improves the quality and stability of the software.

This requires a big change in company culture: to lead the way to another important change for DevOps, which, of course, must have complete management approval to be really effective. The big advantage is that this assures improvement in the quality of the software.

The Operational People Must Be Treated As "First-Class Citizens"

When we design a new feature, the development and the architecture teams must be involved with the operations team. This is because those responsible for the correct functioning of the live software make up the operations team.

The role of the operations staff during architectural decision making is particularly important in the release of a new feature. Imagine, for example, that we must design a new feature for our system. The developer proposes a fancy new component for the web interface and offers a mini demo. Of course, on the developer's laptop, no issues occur. Problems can arise, however, when the component is tested on the actual server. The only people who can respond effectively to such problems are the operations technicians, those who know the server on which the software should be installed and run and who know all the policies related to security, software version, etc. In this case, the operations team can reject the component, because, for example, it does not meet company standards, or the team can start a process to test the server and ready it for the new component.

This means, for this Sprint of the next n-Sprint phase, the component cannot be used, but the operations team can advise the development team when the server is ready. Another important reason for including the operations team in the design of software is log level. For a developer, the message can be clear, but this is because he/she knows the software and understands what's happening. Operations personnel must be able to understand an issue primarily by reading the log. If the log is too chatty or otherwise unclear, this will impede a correct analysis of the error and cause a delay in finding a resolution and identifying a root cause of the problem.

Continuous Integration and Continuous Delivery Policies Must Be Built

Using CI/CD policies helps the development and operations teams to identify faster potential issues with the software. When we establish a practice for CI/CD, we receive constant feedback. An important part of every CI/CD system is the code review. When a developer completes the code, it must be fully tested. First, to ensure successful integration, the developer must ask other software engineers to review the code and raise any issues found in it. This is a very important procedure for the CI/CD system. For every simple update in the software, the developer must begin to adopt test-driven development (TDD) practices. In this way, every commit can be fully tested by the CI software, for example, Jenkins, and promoted to the next step, for example, creation and delivery to the quality assurance (QA) environment for testing.

Having a CI/CD system in place helps to improve the quality and stability of the system. The reason for this is connected to the nature of the system. The entire life cycle of the software is extended every time we commit a new file to it. This means, in the event of a new bug, that we can determine in real time the cause of the error, and we can easily roll back the software delivery. At the same time, we gain the ability to review the software constantly. This helps the operations team to identify security risks and take all the necessary measures needed to eliminate them.

But to achieve true success and avoid creating problems and destabilizing the system, the software engineer must invest more time in unit testing and automation testing. In addition, we must invest time in code review. If we don't have a good test coverage of the software, or don't really review the code, we can release software that is potentially dangerous and compromise the security of the application.

Barriers to the IT Department Must Be Removed

Normally, development and operations teams use different software for their daily work. This can create barriers that must be removed, if we want to ensure an effective DevOps journey.

DevOps can unify communications tools and promote communication across different IT-related departments. In this way, we can coordinate the time frame for the software release and can better plan the work involved. If, for example, a new feature introduced to the CI system creates a security bug, the security team can use the common channel to communicate to the development team, and this can put in place a fix for solving the problem. At the same time, the operations team can be advised of every step the developer has taken with the software and can be ready for the release time.

The Release Process Must Be Automated

Analyzing the error rate, we can positively identify humans as the main cause of failure. The main focus of DevOps is to reduce human and other errors and reduce the time to market. To achieve this, one of the important changes we must make is to automate the release process.

With an automatic process for releasing the software, we reduce the human interaction with the system. Less human interaction reduces the number of failures, because we have a predictable and repeatable process in place.

The other advantage of having an automatic process in place is the possibility of defining the IaC. With the IaC, we can define, via code, what type of structure we want for our software. In addition, defining the IaC makes the infrastructure predictable and allows for the faster release of upgrades. Automating the release process and defining the infrastructure reduces or removes, in the best-case scenario, human interaction and, for this reason, effectively reduces the error rate.

Agile Practices Must Be Promoted Across the Entire Company

DevOps was born during an Agile conference, and to be effective, a company must begin to implement Agile across all its departments. Normally, the Agile practice is mostly used by the development team, but for a good DevOps journey, we must spread this practice as well to the infrastructure and operations teams. If the entire team works in Sprint and, if possible, shares the same backlog, this can help to improve communication. During the Sprint planning, the other team can adjust the work in order for it to be more effective.

At the same time, with Agile in place, we can improve communication and make the work more visible across the team. At the end of the Sprint phase, we can see a demo of the work of the other teams. This helps to see how effective the work is and how to improve the iteration from one team to another.

Reasons for Adopting DevOps

There are different reasons why a company decides to adopt DevOps. Normally, the adoption of the DevOps philosophy is related to improvement in the quality of the software and a better way of managing its release.

When a company adopts DevOps, the first step is to improve communication across teams. This characteristic of DevOps is shared by the Agile methodologies and can be put in place only with a harmonization of the tools used across the company.

This change is not always easily accepted by all IT employees. The initial resistance is usually to the change of culture necessary to adopt DevOps. In general, the life cycle for designing and implementing infrastructure is managed using the ITIL. This means the procedure follows Waterfall methodologies, because it is essentially impossible to configure a server without the server being physically in your hands.

Adopting DevOps means changing the way we think of infrastructure: where possible, migrating it to the cloud, adopting infrastructure as code, and adopting the compatibility of the case, using Sprint to manage the work. This demands that all teams use common project methodologies and create a common product backlog that is shared with the development team, in particular, when the project involves new infrastructure.

Another reason for adopting DevOps practices is the improvement in the quality of the software released. With DevOps, we can adopt some procedure for improving the quality of the software. For this we must have in place continuous integration and continuous delivery. With these, it is easy to identify errors when we push the code on the repository. In addition, because we have continuous delivery, we can release the software directly on the QA more times per day. This ensures a continuous check of the software and continuous feedback for the software engineer.

These are just some common reasons that drive a DevOps journey. Whatever the reason, it is important to understand what actors are involved in DevOps. To do that, we must clarify some misunderstandings commonly connected with the use of DevOps. We must try at this point to identify the common mistakes associated with DevOps and to clarify its role and who is involved.

What and Who Are Involved in DevOps?

In talking about DevOps, we can encounter some misunderstanding of what it is and who is involved in it. The first myth regarding the adoption of DevOps is associated with the professionals who deal with it. DevOps, for many people, involves only software engineers, system engineers, and system administrators.

This assumption is incorrect. When a company decides to adopt DevOps, the first change required is to improve communication across the various teams. This means not only development and operations but other teams as well, such as QA, security, and business. To be effective and successful, a DevOps journey requires that all team members work together. The goal of DevOps is to reduce time to market. This means that when a new feature is designed, every team must communicate, to reach the goal. The QA engineer must respond quickly to the software engineering team and communicate any glitch found in the software. At the same time, the software engineer must communicate with the security team, to describe what the software does and what libraries are used, and to allow the security team to martial the necessary assets to ensure the safety of the software. The business analyst must to be aligned with the software architect, and the software engineer with what the customer wants.

As you can see, to undertake a successful DevOps journey, the whole organization should be involved. Every team must take responsibility for a small part of the business, but in tandem with other teams. DevOps seeks to remove communication barriers across teams, making it easier to identify and correct errors during development and not after release. This ensures better software, a more timely release to market, and better alignment with what the customer needs and wants.

All these actors must work together like musicians in an orchestra. If all respect the symphony, everything runs smoothly, but if one team starts to make problems or doesn't practice good communication, the intended goal will be compromised. For this reason, the most important job when adopting DevOps is to improve the coordination of the internal and external teams.

Changing the Coordination

By adopting DevOps, one of the goals we want to achieve is the reduction of coordination. This means ensuring that those responsible for managing the team invest less time coordinating the different operations. This becomes very important when moving the software from the development server to the stage server. If a CI/CD practice is in place, the software is automatically promoted.

When more automatic processes are introduced, human interaction is reduced and, thereby, the requirement of coordination. This is necessary to reduce time to market. Fewer humans require approval; therefore, fewer delays occur. This requires a change in the classic coordination processes. When we adopt a nonautomatic process, normally, when we finish the software development, the team responsible for the development communicates the completion of the development and then coordinates with the other teams responsible, to move the software onstage. This coordination essentially delegates to humans different ways in which to communicate, for example, via e-mail. DevOps tries to change this way of coordination, reducing the human interaction and, of course, changing the way coordination is actualized.

The coordination has different means of being actualized. These change depending on the context—whether the team is remote, on-site, or partially remote. The normal attributes necessary for good coordination are to be

- Direct
- Indirect
- Persistent

These three attributes define how we manage coordination across the team. Every style has its strengths and weaknesses, so we must be sure to use the correct type of coordination for our purposes. The wrong type can result in an unnecessary consumption of resources and poor coordination. I will now discuss the different styles and when to use one instead of another. I will describe how to use Agile, to improve coordination and to split it across roles and artifacts of the Agile methodologies.

The goal for all the kinds of coordination is to improve communication and, with that, reduce time to market. Remember: The ultimate goal of DevOps is to reduce time to market.

Direct Coordination

With direct coordination, those responsible for the coordination know each other. This means the coordinator directly coordinates the job of every team member. This kind of coordination requires a lot of work from those responsible for the coordination. Normally, this effort can be mitigated when the team is managed using Scrum. In this way, during the stand-up, the staff responsible for the coordination can receive direct feedback regarding the status of the team and take decisions about that.

Indirect Coordination

By this type of coordination, we don't just coordinate people, we coordinate a team, for example, system administration, software engineering, etc. This kind of coordination requires greater coordination, because we don't really go deeply into the details of the task but approach it from a higher level. Imagine, for example, that we must manage new software being put in place, a new piece of infrastructure, and new software functionality. The coordination we want to have is not about detailed tasks but a general view of the status of some specific task. This view gives to the coordinator the capacity to have a plan and start to move on the other activities to calculate an estimated time for the release.

This kind of planning is normally delegated to the product owner. Keeping still to the Scrum style of management, the product owner doesn't really go deep inside a single functionality, but he/she takes an overall view. The production owner is responsible for the entire project and, of course, can help the team reach the best result, reducing unnecessary effort.

Persistent Coordination

This is not really a kind of coordination but essentially an artifact. Persistent coordination refers to all the reports and e-mails sent when a decision is reached about a project.

Persistence gives the team all the instruments for keeping a daily record of the production story and allows teams to make new decisions, based on the history of the project and prevents any misunderstandings about the project itself.

The DevOps Chain

Until now, I have discussed only the kinds of coordination available and what kinds can be used to improve communication. However, the most important question we want to answer is why coordination is so important in DevOps.

The reason is simple. The DevOps movement progresses according to a "toolchain." Essentially, this toolchain is used to define every step of the production process.

Figure 1-1 shows the phases for the DevOps of a software release. Every phase can be managed by a different team. For this reason, strong and clear coordination and communication are important.

Figure 1-1. *Porter's value chain for DevOps*

To better understand the importance of coordination and communication, we must understand how every phase is connected to the other, creating a "chain" of production for the software.

The first phase is the *code*. During this phase, code for the software is created. Every developer puts the code in a common repository, for example, Git, and this leads to the next link in the chain.

The second phase is *build*. This phase is directly connected with the continuous integration practice. The code previously committed is downloaded in the build server and then built in an automatic way. At the same time, a test is performed for the first time. If all elements of the test are successful, the next phase begins.

The third phase is the *test* phase. The software previously built is tested by some automatic process, but this time, the software is tested altogether. In the build phase, only the unit test connected with the specific functionality we release is executed. If the system doesn't find any issue, the software is promoted to the next phase. In case of failure, the software will be rejected, and an automatic system will advise the developer of that.

The fourth phase is the *configure*. This phase requires a clear distinction. When we have good and tested DevOps practices in place, we can have continuous release. This means, obviously, the continuous release of the software in production. However, for software that is mission critical, this phase is normally split into two different parts. The first release is intended for a restricted number of servers called *canary servers*.

Note The term *canary server* designates an intentionally restricted number of servers that are used to test the new software. The purpose of the canary server is to allow a real user to use the new software and provide real-time feedback about potential bugs and the quality of the server. A canary server is very important when we want to be sure not to release software that can be potentially destructive to the company. At the same time, canary software can be used for pen testing and to improve system security.

The fifth phase of the chain is *release*. In this phase, the server is configured, as well as the infrastructure for the new software. This phase defines IaC. The server is created and managed using the code. Such software as Chef, Puppet, Ansible, and CloudFormation are examples of software for creating IaC.

Note Infrastructure as code (IaC) is at the core of DevOps. IaC provides the ability to create the infrastructure for a new server. This guarantees the integrity of every new release, because human interaction is reduced and the integrity of the release is improved. In addition, IaC allows DevOps to create a different environment as a request. This makes it possible for the developer to create a different environment and different test environments directly on request. With IaC in place, we can create and orchestrate an *immutable infrastructure*, that is, an infrastructure composed of some immutable components that are replaced every time we release the infrastructure. Instead of updating a component for our infrastructure, we can simply deploy a new immutable component with the necessary update. This guarantees the stability of the entire infrastructure and ensures that an infrastructure always yields the same result with every release.

The sixth phase of the chain is *monitor*. This is extremely important for providing continuous feedback about our software and infrastructure. Monitoring is very important in DevOps, because it allows the developer to gain feedback about the software, including an average of the failure, the kind of failure, etc., and, at the same time, can be used to check the metrics of the server and provide feedback for autoscaling it.

Coordination and communication are crucial for putting the complete DevOps chain in place. This is because every phase requires a good coordination at every step. We must ensure reliable feedback at every step, because we must react quickly to errors and adjust the system, to prevent new errors.

Defining the Development Pipeline

To ensure a successful DevOps journey, one of the most important jobs is to define the development pipeline. Building this pipeline is essentially the core of the changes required by DevOps.

The first of the changes in the development life cycle is to put in place continuous integration. This requires some changes to our development practice, which can be summarized as follows:

- Define the unit test.
- Define a branch policy.
- Have in place a continuous integration system.

These three practices are the backbone of the development pipeline. The first, the *unit test* occurs every time the developer commits code to a central repository.

When the code is committed to the software for continuous integration—for example, Jenkins—this compiles the code and executes the unit test associated with the software. In the case of failure, an e-mail, with the test results, is sent to the developer.

Because we don't want to break the main branch, we adopt the second practice, the *branch policy*. This is important for maintaining a clean master branch. When a development team adopts this policy, every developer creates a specific branch when developing a feature. This policy is strictly connected to a code review. For every merge with the master, a code review occurs, after the build has been completed and correctly tested. Essentially, for every commit, only the branch is built. In this way, in case of error, the master is not broken and is always ready for a release.

In the case of a positive build, we can ask for a code review, and when the code review is complete, merge the branch with the master, and, of course, restart a complete system for *continuous integration*. With continuous integration in place, we build and test every time we commit into the master or a branch.

Continuous integration must be paired with a good communication system. In particular, we must have a good mail system, to send e-mail to the developer to brake the continuity in the pipeline.

With this pipeline in place, we have continuous software production. What closes the pipeline are the release and monitoring.

During the development life cycle, the release does not happen during production but in QA and testing the server. This release happens automatically. Essentially, it is a promotion of the software built by the continuous integration system. This release is used for testing purposes by the QA engineer, to test the software, provide faster feedback to the developer, and put in place fixes to any bugs faster.

Having a release in QA is important, not only to fix bugs, but to start the monitoring phase. Monitoring is very important in DevOps, to reduce and prevent errors for occurring in the system.

Monitoring is very important for checking and maintaining the stability of the system. A good monitoring system must check not only the availability of the system, for example, if the network is available or the software is working, but can be used for preventing future errors.

There is a lot of software for monitoring, for example, Nagios, Prometheus, Zabbix, or the ELK combination, Elasticsearch, Logstash, and Kibana. All this software has its specific strengths and can be used in combination for achieving the best results.

One of the crucial reasons for effective monitoring is the log. With a good log, it is easy to initiate some log analysis policy. This policy is intended to isolate common error conditions and define some practice of mitigating the error and, at the same time, give to the developer the critical space to fix the software.

Centralizing the Building Server

Centralizing the building server is crucial for building the correct pipeline. When we design a DevOps architecture, we must think of reducing points of failure.

When we adopt a build server, we centralize everything in one server. This means that we use a different software to release our new software. Having only one server, or cluster, for building new software means that there is only one point of failure. Any problem is centralized at only one point. This reduces the cost of maintaining the software and speeds up the release process.

The building server is normally connected with an artifact repository server. The repository server is where the build is stored and saved. It is associated with continuous release. Essentially, with this practice, we build the software every time and release it to a server. This server essentially maintains the different versions of the software we build on the server. Normally, we establish a naming policy to maintain the different software versions. This is because we want to identify every version of the software uniquely.

With the artifact server, we can easily centralize one point for release of the software. In this way, we can have different versions of the same software, and, if we use Docker, we can have different versions on the same server at the same time. We can also start them at the same time, with some minor adjustments. This allows the QA engineer, for

example, to undertake some regression test, and in case of new errors, identify exactly what version has the bug. This allows the developer to understand exactly what change to the code introduced the error.

Monitoring Best Practices

To be effective, monitoring must be combined with some other practice. A log analysis is the most important practice for preventing errors and understanding how the system functions. Some software is required for analyzing the log and making related predictions.

The software most commonly used is ELK (Elasticsearch, Logstash, and Kibana). This ecosystem is helpful because it gives a complete log analysis system, not only providing alerts, but also a graphical representation of the error and the log.

Log analysis is very important for improving the quality of software. One important practice we can put in place is to have some software that not only identifies the number of errors but graphs these as well.

Having a graphical representation of the errors is important for providing visible feedback about the software, without the necessity of reading a log, in order to understand the status of the software.

Monitoring is the backbone for every DevOps practice, and if we want a very successful journey, we must be sure to have a good monitoring system. At the same time, we must start to monitor not only the production but possibly the canary server. This is because it can reveal an error, and we can solve it before release to production. Monitoring can take two forms. *Black-box monitoring* tests a piece of code as if it were in a black box. It reveals only the status of the system, to determine whether it is alive. It doesn't really indicate what is happening inside, because the monitoring is external. An example of black-box software monitoring is Nagios.

The opposite of this is *white-box monitoring*. This type of monitoring provides a clear picture of the inside of the system, such as, for example, the number of HTTP connections open, the number of errors, etc. Prometheus is an example of white-box monitoring software.

Best Practices for Operations

In DevOps, the operations team has a big influence on achieving the best results. The importance of the operations team is strictly connected to the quality of the software and what the customer thinks about the company.

In case of an error, the operations team is the first face of the company. This team is normally delegated to maintain the software in the production environment.

The only point of contact with the software is the log. For this reason, some member of the operations team must be included when software is designed, and, more important is the feedback they can provide when software is released for testing. This is because if the log is insufficient, the operations team can't really identify the error, which means more time will be required to fix the issue.

At the same time, the operations team can help to identify common issues and provide documentation to solve them faster. This documentation is actually a step toward resolving the issue. It is used essentially by first-line operations engineers. It is "live," meaning that it is never closed and must be carefully managed, so that it aligns with the most recent software updates.

The documentation must indicate common errors in the log and show how to solve the root cause(s) of the problem. This documentation should be written with people who don't know the system in mind and, based on that, must provide specific details about the appropriate steps to be taken.

Another operations practice we can put in place is *developer on-call.* This practice introduces a new figure to the operations world. The developer on-call is essentially a software engineer, working with the operations professionals to resolve errors in production. This has two principal advantages. The first is a reduction in the time it takes to identify and fix an issue. Because one of the developers works on the issue, he/she can easily identify what's gone wrong and what part of the code creates the issue. This can drive the operations team's efforts to fix it.

The second advantage is improving the level of responsibility. Because the developer works to fix a live issue, he/she understands better what's wrong with the software and thus can improve the way he/she writes the software and the log, because a bad log can result in more work for him/her in future.

Conclusion

In this chapter, I have offered a brief introduction to DevOps—what it is and how a movement was born. DevOps is very important in relation to the cloud. Cloud development requires software that is always live and designed to be released faster and of higher quality.

DevOps puts the accent on quality and on time to market. It allows for a simple design microservice architecture, because of the practices connected with continuous integration and delivery, which help to deliver faster service on a system.

DevOps is very important to modern software development, and more companies are starting to adopt it, because it promotes some best practices necessary for improving the quality of software. DevOps requires a change, not only in how we think of infrastructure, but in how well we design and organize the internal company infrastructure.

DevOps essentially represents a change in corporate culture. To ensure its optimal practice, it is important to change the organization, so that its priorities align with the requirements of cultural change, practices for success, and change that is driven by management and, of course, approved by the engineers.