DevOps

Introduction to 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 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.

Adopting DevOps

- 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

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

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

- 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.

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

- Reasons for Adopting DevOps
  - What and Who Are Involved in DevOps?
  - Changing the Coordination
- The DevOps Chain

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

- 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*.

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.

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

Defining the Development Pipeline

Centralizing the Building Server

Monitoring Best Practices

Best Practices for Operations

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.

Introduction to Continuous Integration and Delivery

- Continuous integration (CI) and continuous delivery (CD) grow in popularity every day. This is because they are crucial to reducing time to market and improving the quality of software.
- With the practice of CI and CD, every time we release software from a central repository, it is built and released to test. This represents hundreds of deliveries every day.
- CI and CD are strictly connected, and one is an extension of the other. Both practices have some associated costs and savings.
- In this chapter, I introduce CI and CD and try to show how they are important to our DevOps journey

Definition of Continuous Integration

- The definition of CI is quite simple. It is a development practice that requires the developer to integrate code in a central shared repository. Every time the developer commits the code, it is integrated with other code and verified by the execution of a test.
    - CI starts every time we commit code to the centrally shared repository. This means that every time we change something, for example, a label on an HTML page, or a variable, we test the entire solution, because we test the solution at every single commit. We can find errors more quickly and easily in the build and fix them. What we do at every commit is essentially build the entire solution.

    - If we skimp on any one of the three variables, we can reduce the quality of the software. In a CI scenario, all CI processes start with the commit of the source code in the repository.

- A CI scenario can be designed with these simple steps:

1. The developer commits the code in the repository.
2. The CI server pools the repository, downloads the last code, and starts to test. If all tests are passed, the server compiles it.
3. The CI server sends a notification, via e-mail, slack, etc., with feedback about the integration.
4. The CI server continues to pool the repository, to check the new change.

- Figure 3-2 shows a sample CI system. Here, we can see that we have a mail server that is used to send the feedback to the developer.
- Feedback is crucial to a good CI system, because it provides an immediate critique of the build, and the developer can use it to resolve any issues faster.

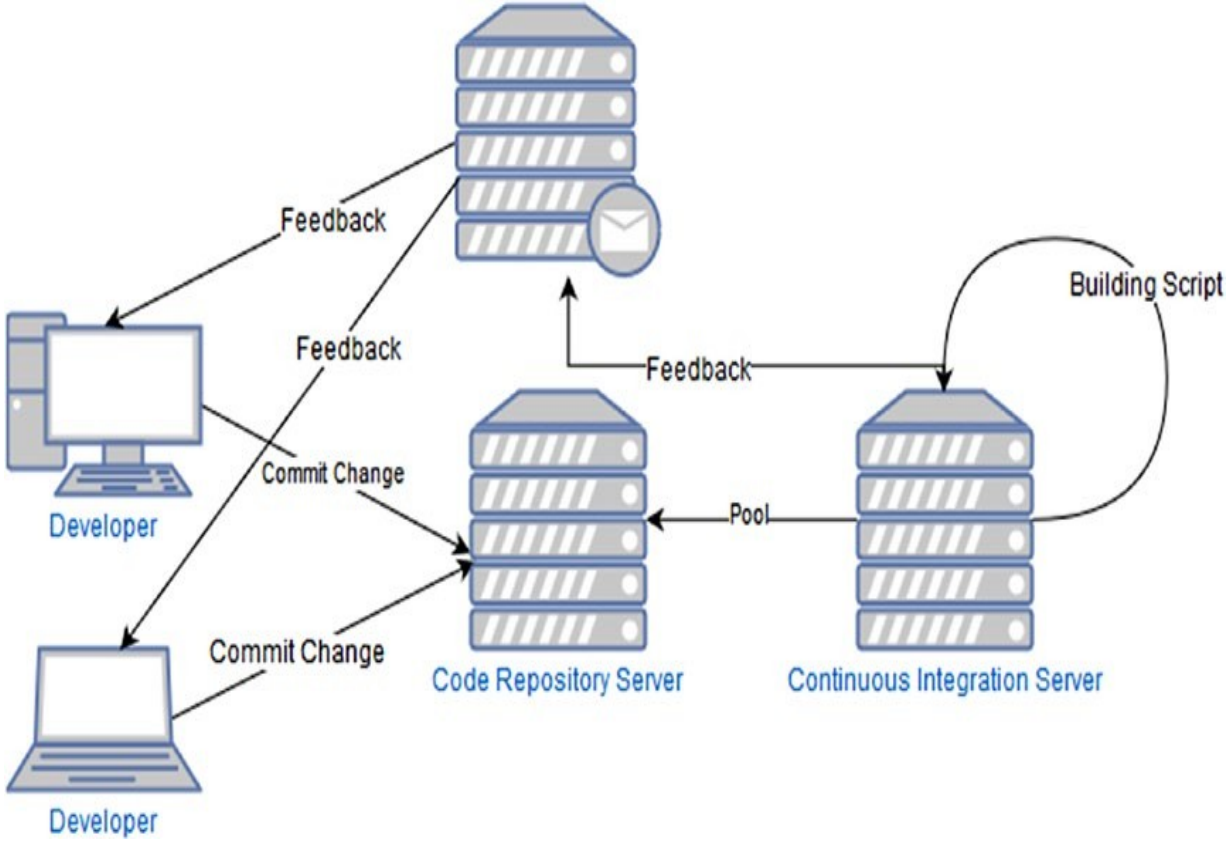


Figure 3-2. *A CI system*

The Code Repository Server

- The code repository server is where we store our software. This is essentially a software for the repo, like Git or SVN.
- The server can be in-house, meaning that we have an internal server, or external, in which case, we don't manage the server directly, for example, when we put the code in Bitbucket.
  - A good CI system must have a repository server.
  - This is essentially the starting point of our process.
  - Every time the developer commits, we start the process.
  - We can have many branches in our repo, but only one master branch, which is essentially where we integrate the other branches every time.

The Continuous Integration Server

•The continuous integration server is responsible for running the integration script every time we commit the code. We can use different software for doing that, for example, Jenkins, Travis CI, TeamCity, etc.

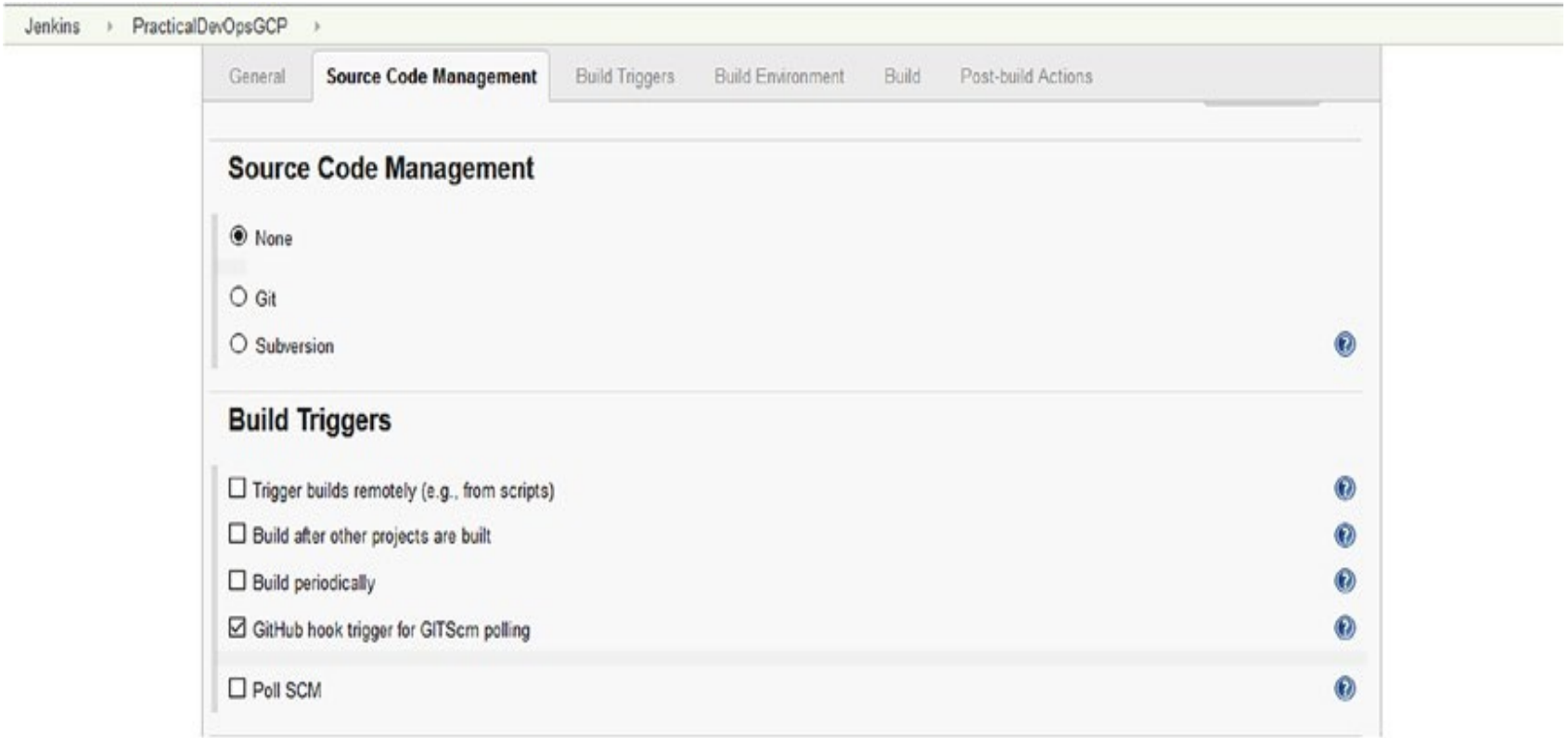
- A CI server executes some specific operations.

1. Retrieves the code from the repository server
2. Merges the last commit with the old software
3. Executes the test on the software
4. Builds the software
5. Sends a feedback with the result

- It is not necessary to have a CI server. We can perform this operation with a simple script, such as Bash, Ant, Maven, or Makefile. We can write a simple script to merge and build the software, such as the following:

```
#!/bin/bash
function integrate_code() { SOURCE=$1
DEST=$2

git checkout $DEST

git pull --ff-only origin $DEST result=$?

    if [ $result -ne 0] then
        echo "Error in pull" exit 1
    fi

    git merge --ff-only $SOURCE --no-edit result=$?
    if [ $result -ne 0] then
        echo "Error in merge" exit 1
    fi

    git push origin $DEST result=$?
    if [ $result -ne 0] then
        echo "Error in a push" exit 1
    fi return 0
}
```

- This script merges the code from a branch with another. It is a very simple script and just a piece of a more complex building system.
- When we use a CI server, we can reduce the number of scripts we need to maintain, because a CI server starts a build in an automatic way. For example, we can configure Jenkins to start a build in different ways (Figure 3-3).



- You can see from the preceding figure that we can connect Jenkins with most of the source control management systems, such as Git, Mercurial, etc., and we can trigger the build with a different option, for example, GitHub hook.

- In this way, when we commit the software in Git, Jenkins automatically starts a build. By adding to the automatic build, we can build at a certain time or use an external script to start the build.

- In addition, with a CI server, we have a dashboard from which we can see what builds are good and what builds failed. This can offer an immediate visual status report on our software.

Continuous Delivery

- CD is a software engineering practice used to release software within a short cycle. This means that with every build, we create a new build of the entire software. This does not mean that we release the software to production, but if we want, we can release it. This is the difference between continuous delivery and continuous deployment.
    - With CD practices, we always have a build ready to use. This allows the QA team to start testing immediately, with a restricted number of features, and give immediate
- feedback to the development team. This reduces the time to fix the problem and improve the quality of the software itself. Of course, this depends more on the environment. In most of the environments with a CI/CD system in place, we don’t really need a QA team to execute the test.
    - This type of approach helps to reduce costs. Maintaining the software or resolving an issue during the development life cycle is certainly more efficient than fixing a problem when the software has been produced. In addition, with CD, we always test a small part of the software, because CD takes place with every commit, thus reducing the risk of releasing software with a destructive bug, for example.

- The idea behind CD is similar to that informing DevOps, but they are two different practices. DevOps concentrates more on changing an entire company culture. Instead, CD concentrates on producing a new software build. However, because DevOps essentially represents a change of culture, CD and CI practices fall within its sphere.

- CD (see Figure 3-4) is an extension of CI, because CD adds another step to CI. For this reason, if we want to have good CD in place, we must have a strong foundation in CI.

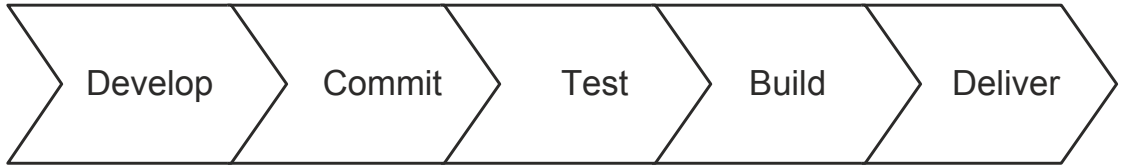


Figure 3-4. *The continuous d chain*

Differences Between Continuous Integration and Continuous Delivery

- CI and CD are similar, but there are some differences between these practices. CI concentrates on integrating the software with every commit. This occurs after unit testing.
  - CD extends CI, because it adds another layer after integration and testing of the software are complete. CD builds the software and prepares it for potential release.
  - CI places a big emphasis on the testing phase. This is very important for CI, in particular when code is merged with the main branch. The goal of CI is not to lose functionality after the merge.
  - On the other hand, CD places great emphasis on building software. With CD in place, we can decide to release new software on a daily basis. In 2011, Amazon had an average release of new software every 11.6 seconds. This is a huge number of releases per day. With continuous release, we automate any step and process required for achieving this result.

Strategies for Continuous Delivery

•To ensure good CD, we must have the following:

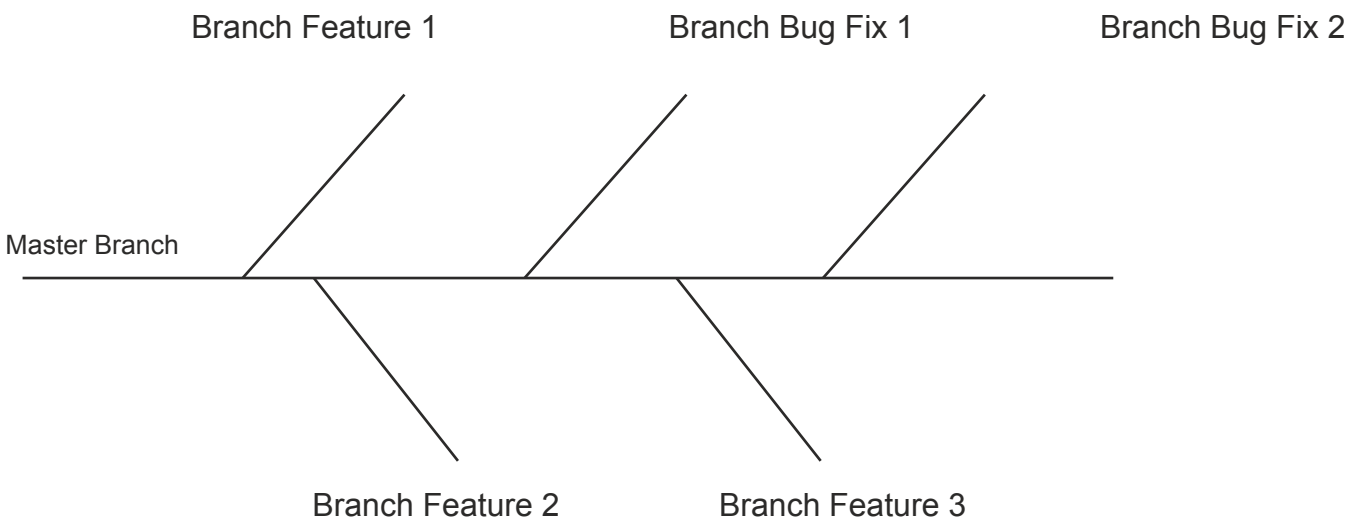
- Good branching strategies
- A strong unit test policy
- An automatic testing phase
- Automatic code promotion

- All of the preceding practices are strictly connected and help to produce good and strong CD. Some of them are connected to CI, such as branching strategies and unit testing; others are more connected to CD.

Good Branching Strategies

•In CI, the goal is to integrate the software with the main branch. With that in mind, we can develop our branching strategies.

- The most common branching strategies (Figure 3-5) are to create a branch for every feature/bug we work on. In this way, we can merge single features with the master branch.



- Because we have a different branch for every feature of a bug, we can play with the code without breaking the master line of code.
- This means that during development, we always have a buildable and potentially releasable line of code.
- When we release the software in our branch, we execute the unit test against our branch.
- We don’t test only the feature we develop, but the entire system.
- In this way, we can have immediate feedback about any error we introduce in the code.
- If all tests are passed and the feature is green-lighted, we can start to integrate our branch with the master line of code. When we merge, we start another set of tests.
- We can also start some code analysis, and if it is green-lighted, we can release a new build with the new feature.

A Strong Unit Test Policy

- To be effective, good CD must have strong CI in place, and for strong CI, we must have strong unit testing strategies in place.
- Unit testing is essential if we want to build a good CI system, because testing can identify an error in what we intend to release.
  - Unit testing is important not only for identifying the error but because it can be used to validate business requirements.
  - A unit test must be written before development. This means that we must write the code for passing the unit test. This technique is called
- test driven development (TDD). With TDD, we write the test based on the business requirements and then start to write the code.
- This ensures a correct correlation between the requirements and the code we release.

TDD is normally connected with a code coverage value. This means that we make sure that a certain percentage of code is covered by the test.

A good percentage of code coverage is about 85%.

This essentially covers all the code, and we can be quite sure of the quality of our code with this percentage tested.

Another important practice for tests is the *test pyramid*.

- This phrase is a metaphor used to describe the different granularities of tests in a bucket.

The concept was defined by Mike Cohn in the book *Succeeding with Agile*.

When we think of a test pyramid, we must include three types of tests.

Unit tests (the base of the pyramid)

Service tests (the middle)

UI tests (the top of the pyramid)

- This pyramid helps to test all the important aspects of the software. It is important to have a test pyramid in place. This is because it helps to catch most of the errors and the design of a more reliable system.

An Automatic Testing Phase

- Testing is very important to guarantee the quality of the code. In addition to a unit test, we can conduct another type of test.
- Usually, we have an *integration test*, for check if all the software is correctly integrated with the other components of the system.
- We can add an *acceptance test*. This kind of test is designed to be executed on the entire system.
  - When we execute the integration test, we essentially remove all the mock parts of the test and use the real system instead.
  - Normally, we create mock parts in the unit test phase, because we don’t yet have any piece of the system ready for testing. For this reason, we create a fake response for that.

- The integration test is important for testing the entire system and to validate our integration. In case of any errors in the integration test, we must revert the integration of the code.

- The acceptance test is important for reducing the risk of accidentally removing features and having a build that does not align with business requirements.

- Usually, the acceptance test is designed by the QA engineering team and is conceived to test any integration with the system.

- This test normally tests the UI/UX of the system, although it is not intended to test the software itself but, more generally, the system and the features connected with it.

Automatic Code Promotion

- Code promotion is the basis of continuous release, because it is used to define what version of the software is ready to be released. A code promotion occurs when the test phase is correctly passed and the code builds without any issue.
- Normally, a CI server like Jenkins has the ability to promote the code itself. In general, this is done by tagging the code in a specific way or creating a new branch for release.
- When we promote a release, what we essentially do is release the code in a different kind of server. For example, we move the code from the development server to the staging server.
- The different server can be used, for example, by the QA engineering team, for executing some additional manual test.
- When we have a CI system in place, usually we have a file to define the artifact.
- This file describes all the libraries and the relation of every piece of the software. This is described by the term “artifact immutability” and is exemplified by Maven, with which we can define the system and all its dependencies, to install and build the software.

Code Inspection

- Another important practice connected to CI and CD is *code inspection*, a.k.a. *linting*. This practice is very important for maintaining a good architect level of code.
  - This technique is used to explore the code and create a clear picture of what it looks like. We can, for example, identify if the method is too long or complex.
  - We can use CI to produce a quality code check. With Ruby, for example, we can use RuboCop. This tool analyzes the code and shows all errors identified in it. In Python, we can use PEP8 to enforce some rules. The use of these rules enhances the quality of the software, because all the development follows some specific rules.
  - Another important check on the software is the *cyclomatic complexity*. This is a measure used to determine the complexity of a program. It measures the number of independent linear paths through the method. These are determined by the number and complexity of conditional branches. When we have a low cyclomatic complexity, this means the method is easy to read, understand, and test.

Benefits of Continuous Integration and Continuous Delivery

Until now, I have discussed and presented the differences between CI and CD. Both practices have some costs and benefits (see Table 3-1) that we must consider when we adopt their practice.

- We can see from the preceding table that both practices require some changes.
- These changes are connected to the way the developer writes the code and how it connects to the infrastructure put in place.
- The major cost is essentially creating and maintaining the CI server, because we must configure it for every new feature we add. We can reduce this cost by creating a Jenkinsfile.
- This is a Jenkins feature that allows us to create a pipeline for CD. Using Jenkinsfile, we can automate and store in the repository our process for the pipeline.

Designing a Continuous Integration and Continuous Delivery System

- For putting in place a complete CI and CD system, we must make some changes to our infrastructure and in our architecture. The architecture changes are not directly connected to the software itself but more in the way we produce the software and release it.
  - The first change we must make is in how we write the code. The developer must start to write the unit test for every single class or function we release, but to be really effective, we must use the TDD technique.
  - This is because, otherwise, we risk writing a test to pass the code, and not to test the requirement we want to implement, which can reduce the benefit of CI.
  - Another change we must initiate is to force the developer to integrate the software as soon as possible—at least one time per day.
  - Otherwise, we can spend more time integrating the software and the test phase, and fixing a bug can take a very long time.

- We must also put in place some rules about the code.
- We can implement a code- inspection system, using a tool such as *infer*, developed by Facebook, which can check multiple languages, such as C, C++, and Java, and produce a report indicating a line

- of code with an error. This can help to improve the quality of the software and reduce potential bugs.
- Other tools, such as PEP8 or RuboCop, work with a specific language and are often used to force some rules regarding the complexity of the method, number of operations executed in the method, number of lines of the method or the class,
- and length of the lines of code. These rules help to have a readable and maintainable code.
- This does not directly improve the quality of the code, but it helps to reduce the maintenance time required by the code.

- To create a good system for CI and CD, we must adhere to some principles.
- Commit the code frequently. Every minor update to the code must be committed and tested.
- Don’t break the code with the commit. With the first commit, execute a local build and test, because the code we commit doesn’t stop the cycle for the CI.
- Develop unit tests. Every commit must be associated with a strong unit test, because we must test to validate the code.
- Create a script for building the software automatically. We must reduce human interaction, which means we have to create a script for building the software and ensure this works every time, to give us the consistency we need for our system.
- Build the software for different environments. With a CD system, software development has different stages. Normally, we have
    - a development server, a staging/test server, and one or more production servers. Every environment has different characteristics and, of course, connection parameters. We must create a system to build software in each of these environments.
- Design pipelines for the software release. To improve quality, we must create a script that automatically promotes the software at every stage.

Design a strategy to release the software at every stage. Because different stages are involved, we must design a strategy and an architecture for the software release. For example, when the software is built in development and the test is passed, this must be promoted at this stage. By doing so, we can easily create a Docker image, release in the registry, and, by software for orchestration, release it in the stage.

- These principles are the foundation for a good CI/CD system. Of course, they must be adapted to specific company needs, but, in general, if we follow these principles, we are sure to reduce human interaction and put in place a good CI/CD system

Building Continuous Integration and Continuous Delivery Pipelines

- To build a good system for CI and CD, we must create a pipeline. With a pipeline, we can define the steps necessary for building the software and eventually release it in production.
  - When we build software, we can identify different stages. Every stage is responsible for a specific validation of the software. The basic pipeline is composed of three stages.

1. Development
2. Staging
3. Production

  - When we define a pipeline, we essentially create a system for promoting the software from one state to another, when a certain condition is in place. This process must be managed programmatically, so that it can be easily changed/updated and reduce human interaction. Today, there is a lot of software that we can use for that, for example, GoCD, Travis CI, GitBucket, Circle CI, and Jenkins.

All this software can be used to visually create a pipeline with the different stages we want for our software. Much of it supports some type of scripting language.

Having a script for the pipelines is important, because we can save the script in a software repository.

- If we have to create another environment, we have only to download the script.
- For creating such a script in Jenkins, we use a Jenkinsfile.
- With this file, we can define all the steps we want for our pipeline and, of course, use it to promote the software from one stage to another. An example of a Jenkinsfile (Listing 3-1) follows

•***Listing 3-1.*** A Basic Example of a Jenkinsfile

```
•pipeline {
  • agent any
•
  • stages {
        • stage('Build') { steps {
            • echo 'Building..'
         • }
     • }
        • stage('Test') { steps {
            • echo 'Testing..'
         • }
     • }
        • stage('Deploy') { steps {
            • echo 'Deploying.       '
         • }
     • }
  • }
•}
```

- From the file, we can define different stages and different agents, which is important for our CI/CD system. The different stages can have different parameters for work.

•For example, DB connection of system access passwords. In the stage section, we can prepare our system to get this value and change the behavior for response at different stages. Not every CI system has only three phases, but they are a good starting point.

- Another important piece of our CI/CD system is the feedback system. This is essentially an e-mail sent out to advise the user about the status of the build. This message is very important, because the developer can react and fix any broken part of the build. The e-mail can be very simple.
- It must include only the number of the build, the error that has occurred, and the tests failed. This information helps the developer to identify issues and fix them faster. The system must escalate the e-mail if the n-build failed to CC, for example, the team leader.

Continuous Database Integration

- When we release software, usually we have a database in which we store the data when we change the software. This can be associated with a change in the data structure, for example, a new table. In this context, it is important to establish a continuous database integration commonly known as *database migration*.
  - Continuous database integration is the process of releasing and rebuilding all the database and populating it with new data, every time we release the software.
  - Following this process ensures that we have a database that is always aligned with the last code and, of course, provides a fresh set of test data. Another benefit of this process is that it identifies any issue with the database every time we release the software. To take advantage of this process, we can create and maintain the script for
- the data manipulation language (DML) and data definition language (DDL). This script must be stored in the software repo, as with every other piece of code in the system, and then reviewed by the DBA team for approval or rejection.

To adopt this process, we can follow these simple steps during our build:

1. Drop the entire database. This way, every time we build the software, we have a fresh new database.
2. Maintain the DML and DDL script in the code repo. The script for creating the database must be in the code repo and integrated every time we release the software.
3. Have a stage for re-creating the database. Because we continuously integrate the database, our pipeline must have a stage for creating and maintaining the database.
4. Have a code review for the DML and DDL script. The DBA team must be aware of any change we put in place in the database, so that we can easily identify any relevant issue. In addition, we must have a code review, to ensure that the update doesn't break the database.

- Ensure that the test data is always aligned. Because we can change the database structure, we must align the test data to reflect the changes we have made in the database

Continuous Testing and Inspection

- A principal aim a company wants to achieve by building a CI and CD system is to improve the quality of the software released. To ensure this, a good CI and CD system must have continuous testing and continuous inspection in place.
  - The scope of continuous testing is to create reliable software every time it is released.
- To achieve this, we create different types of tests that can be executed in an automatic way every time we build the software.
  - The first type of test we automate is the unit test. This test must be executed every time we compile and build the software. This test is the first point for testing the release and the quality of the software. The unit test can also check the code coverage of our software. Code coverage is important for understanding what percent of lines of code of our software is covered and then tested.
- There is no specific optimal number for the percentage of code coverage, but good code coverage is considered to be between 80% and 90%.
- What is clear is that to write a good unit test, write to effectively test the functionality and not only to achieve code coverage.
- The reasons for having code coverage in place are essentially two

- It improves the quality of the unit test, ensuring that more code is covered and more bugs are intercepted during the testing phase.
- It allows us to be confident that when we develop a new feature, we are not releasing a new bug into the production environment.
  - The unit test is only the first step in our testing system. Another test we must include in the system is the *integration test*. This type of test is designed to test the software with the real components.
  - This phase of testing occurs after the unit testing and uses real data to execute it.
  - During the unit test, we can have the ability to mock some data. For example, when we must communicate with an external web service, the integration test combines the different components and tests all the software together.

- The last phase of testing is made up of the *acceptance test* or *verification test*. This phase of the test is designed by the QA engineering team to test the system from a user's point of view. This means that in the event of an interface, the test is essentially designed on the interface. The goal of this phase of testing is to verify the user requirements and validate them. At this stage, we have the test pyramid build and can easily use it for testing.
- In addition to the different testing phases, another important phase is the *code inspection*. This phase is a check of the code, using a set of rules to produce a report on the software itself. Code inspection can be split into two different phases.

- *Code review*: This phase is in place before the final integration.
- *Static code analysis*: This phase occurs when we integrate the software.
  - The code review is the first phase of the code inspection. During the code review, the code must be approved by another developer first, to be integrated into the main
- branch. The other members of the team review the code and leave feedback about it.

- The static code analysis is made up of two different phases. The first occurs when the developer executes the local commit. During this phase, the code can be validated by some rules. These rules check, for example, the following:

- The complexity of the method
- Lines for every method
- Number of characters per lines
- Comments on the method or class

- There are more tools available for making this analysis, for example, PEP8 in Python, and different languages can have different types of rules applied to them.

- The other type of analysis we make of the code is the static code analysis. This analysis has the goal of highlighting issues connected directly with the code. There are different tools for doing that, but it is normally executed by automated tools. This is important for identifying potential runtime bugs that can appear in the code and to fix them before release to production. When all the analysis and tests are executed, the code can finally be built and prepared for release.

Preparing the Build for Release

•The last step in a CI/CD system is to prepare the release for the build. This follows some simple rules.

- Identify the code in the repository.
- Create a build report.
- Put the build in a shared location. For most modern software, we can have a nexus for our artifacts that allows us to rebuild the software in every system.

- These basic rules can be used to identify the software ready for the build. A ready build can be released faster into production, or in our QA environment

Identifying the Code in the Repository

- Identifying the code in the repository is important for understanding when we have a production-ready build. We can identify the code in different ways.
  - *Create a label in the repo*: The fastest way to identify the last build code in the repo is simply to identify the code.
  - *Tag the code*: A more complex way of identifying the code is tagging. This means creating a tag in the repo with the value for identifying the version.
  - *Create a branch*: Another way to identify the code is to create a new branch. This is similar to the tagging technique, only we use the branch instead.

- To identify the code, we must create software with a unique name. To create a name, we can use a naming convention such as the following:

```
PracticalDevOpsGP.1.1.0
```

- The naming convention we create uses this syntax: *<feature>.<major release>.<minor release>.<build number>*. When we build, we essentially change only the last version of the number, for example, PracticalDevOpsGP.1.1.1, 1.1.2, etc. As with every other feature, this must be created by the CI/CD system in an automatic way.
-

Creating Build Reports

- Build reports include important information we disseminate after the build. This report must consider staff other than technical personnel, so that what is in the report can be universally understood.
  - For build reports to be effective, they must include the following information:
    - Name of the release
    - Feature released
    - Where to get the release
- The report can take the form of an auto-generated message, produced by the CI/ CD system and sent to a specific list of users every time the release is ready. The field can easily be included during the build. The trickiest part can be the feature released, but data related to this can be gathered simply by connecting the CI system with the system for maintaining and designing the software. For example, by connecting Jenkins with Jira, we can determine what task we are still working to connect. In this way, when Jenkins receives the code for the build, it can include a description of the feature, which can be added to our report.
- The report can be used to identify the feature and, of course, by the QA team, to identify any discrepancy in the feature planned to be released and the one effectively released.

Putting the Build in a Shared Location

- When we finish creating the build, we must share it with other teams. Where we put the build depends on the policy we use to release the software.
  - If, for example, we release a WAR or an MSI file, we can put the software directly in a shared server. If, for example, we want to create a Docker image, this image must be published in an internal registry used to retrieve the last image to build.
  - What we must keep in mind is a very simple concept: the immutability of the build.
- When QA tests a specific build version and validates it, we must release exactly the build used in the QA. The system doesn't have to make another build; it just uses the file passed in QA for release.

Releasing the Build

- Releasing the build is the last phase in our CI/CD system. The build release is not intended to be solely for production but can, for example, be used to restrict the number of servers, specular to the production server, on which some customers can try out new features.

  This type of server is known as a *canary server*.

  - Another important consideration is how we release when we release in a cloud SaaS. We don't want customers to have any interruption in the usage of the software.
  - To do that, we must identify a specific way of releasing the software and ensure the reliability of the software itself.
  - In other cases, we can schedule software maintenance windows. At a specific time, we essentially stop the functionality of our software and release a new version of the software itself.

- To release the software without interrupting functionality, we can release it with some specific procedures. The most commonly used include
  - Blue/green deployment
  - Canary deployment
  - Incremental deployment
- The main goal of these techniques is not to interrupt the functionality of the software and intercept potential problems with the infrastructure and the software as soon as possible.

Blue/Green Deployment

- Blue/green deployment is a technique for releasing software that reduces the risk of downtime. It is called "blue/green" because we release two production environments: one called blue and another called green.
- With blue/green deployment, we have only one live environment. The system for CD releases the new version of the software in the environment that is not live. When the software is ready and tested, it is installed in the other environment and then switched in production. With blue/green deployment, we essentially have two similar environments, and we just switch between the two (see Figure 3-6).

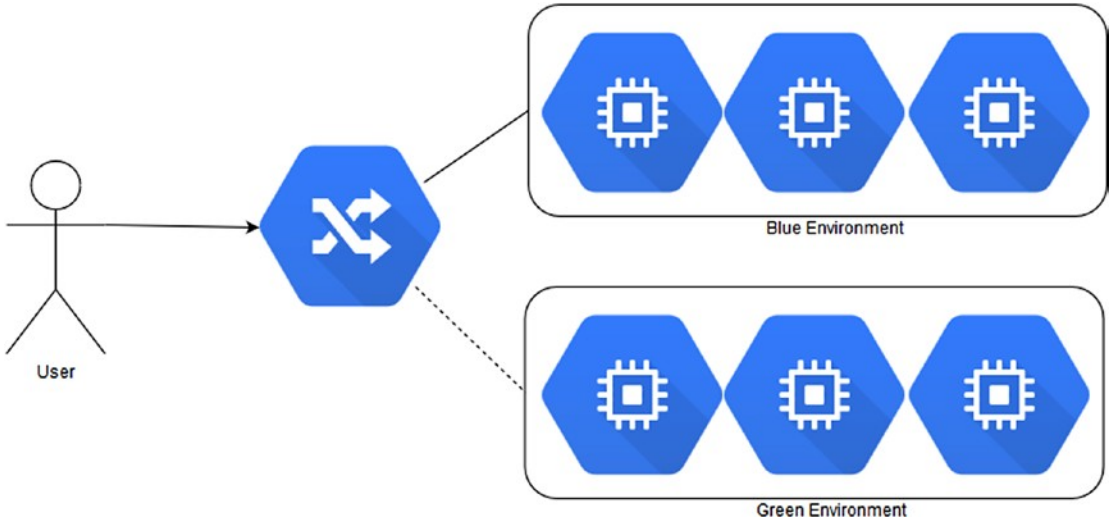


Figure 3-6. *Blue/green deployment*

- Blue/green deployment has some benefits and some costs. With blue/green deployment, we can easily roll back the environment, in case of error, because we always have an environment ready for production.
- The cost is connected with some architectural design that we must bear in mind. The first concerns the database. When we release the software, we may have to modify the table, before proceeding with blue/green deployment.
- First, we must release the database. When we have released the database, we can then switch the environment.
- Another important point we must keep in mind is the user session and other data that can be used by the software.
- We must have a cache common to the environment, in order not to lose this information and allow its use without any issue.

Canary Deployment

- Canary deployment is intended to reduce the risk connected with the release. We release the software in a small part of the infrastructure, which means only a small percent of customers is touched by the release.
- In case of failure, we can easily roll back the release. The release is intended to be incremental in terms of users. We increment the number of users after a certain time, so that we don’t reach 100%
- This type of deployment can be associated with blue/green deployment. The difference is in how we switch the infrastructure.
- We create our new environment, and when we are satisfied, we start to release a subset of the users in the new server.
- Canary deployment is used to provide immediate feedback about the deployment from a restricted number of users.
- This can help to identify and solve an issue without a complete rollback, because we release only to a restricted number of users.
- In the event of a rollback owing to any issue, we can just release to a small number of servers

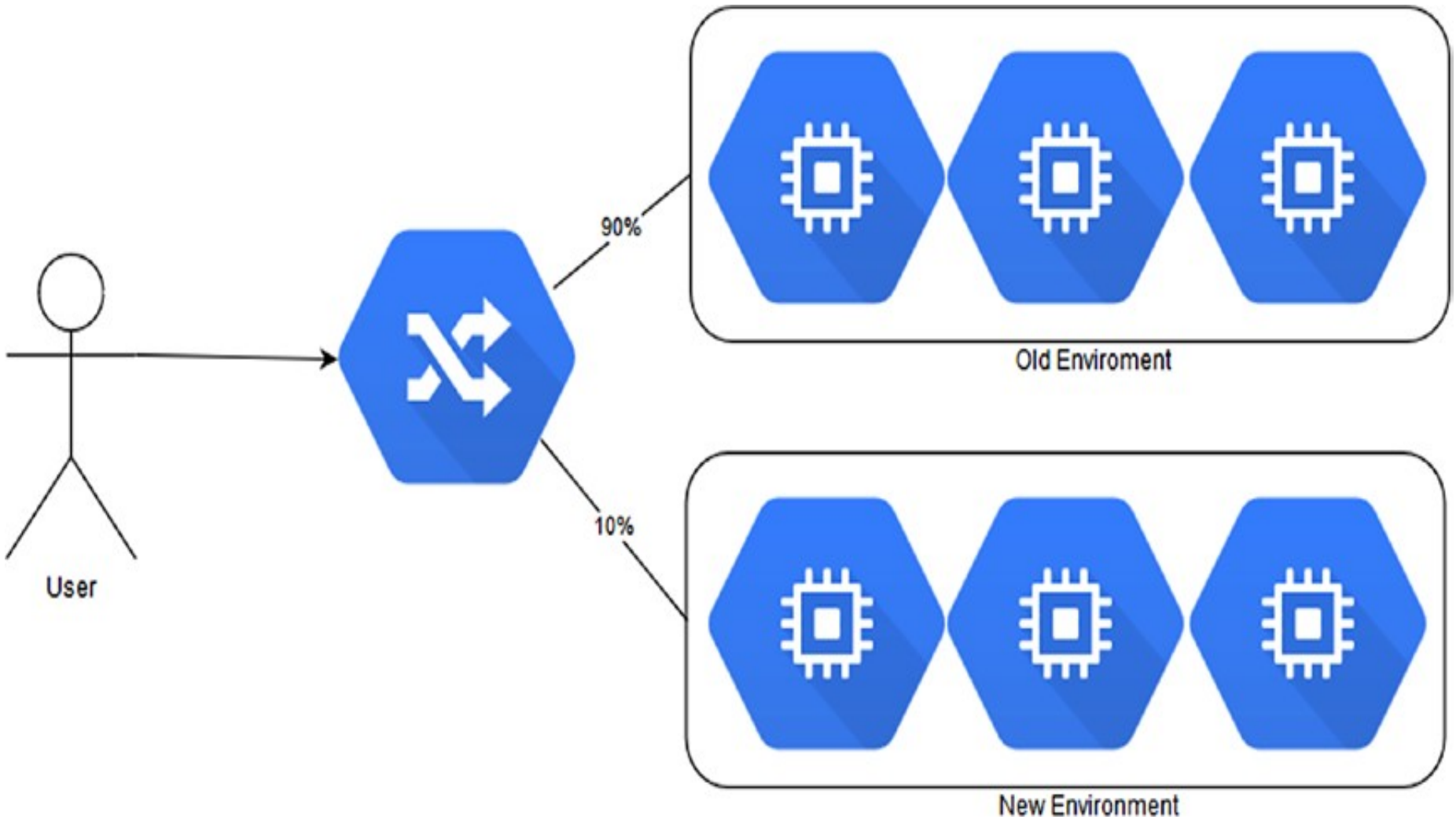
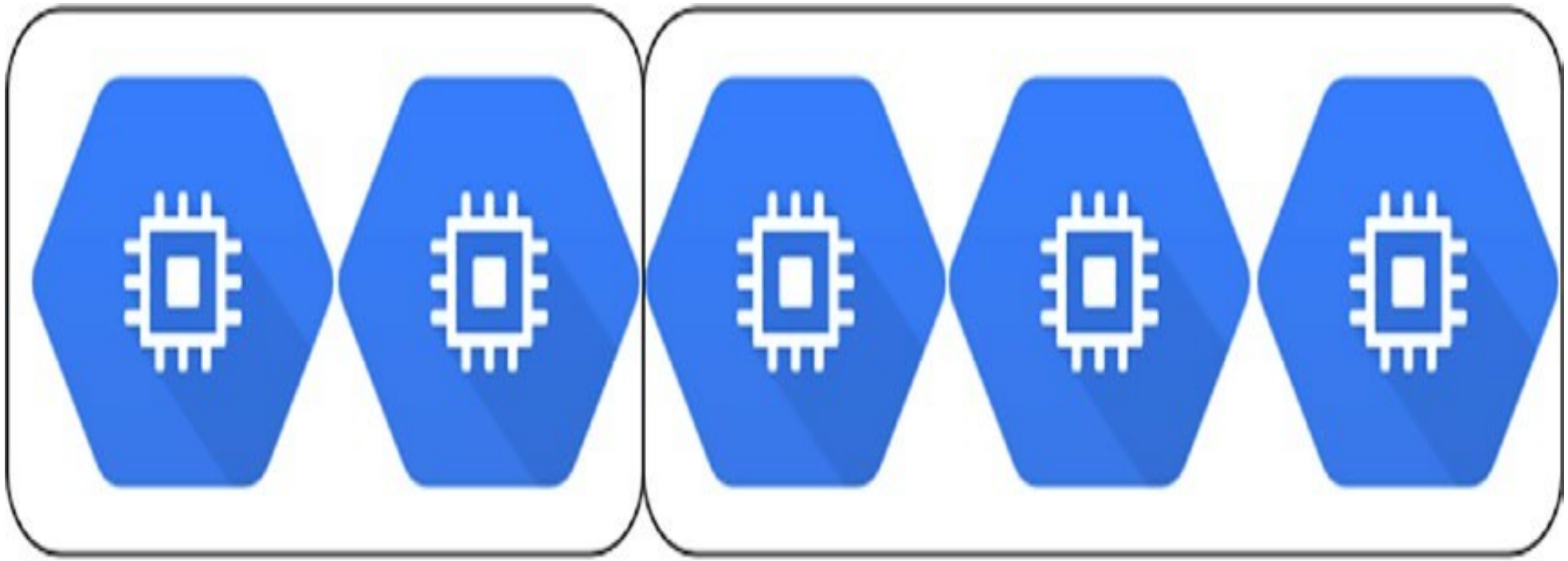


Figure 3-7. *Canary deployment*

- Another benefit of canary deployment is connected to the slow ramp-up in the number of users. When we release new functionality, a slow ramp-up of users is preferred for analyzing the use of the memory and other issues connected with the functionality. At the same time, the allows the chance to create specific monitoring values for the software.

Incremental Deployment

- Incremental deployment is used when we want to have only one hardware in production.
- By this technique, we release to only a percentage of users at a time, for example, to 5% of users. When we are satisfied with the first release, we move to another set of users.
- The incremental deployment process is used with only one line of hardware.
- This is because only a small part of the software is used at a time.
- The benefit of this type of deployment is connected with the small amount of servers we release to.
- Because of this, we can monitor the feature better, immediately identify any issue with the software, and adapt the infrastructure or the code to fix the issue.


New Environment
Old Environment