CHAPTER 3

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.

Adopting CI is cheap. Essentially, we only need a server with Jenkins, and we can start to use it. CI can be summarized in three simple phases (see Figure 3-1).

P. Riti, *Pro DevOps with Google Cloud Platform*, https://doi.org/10.1007/978-1-4842-3897-4_3

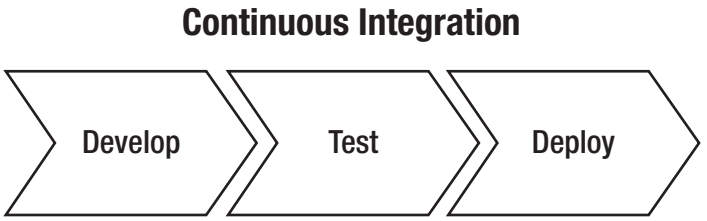

Figure 3-1. *The continuous integration chain*

1. Develop
2. Test
3. Deploy

When we have CI in place, we execute this cycle every time we commit the code in our repository. When we use CI, we create a build every time. This is the essence of CI: we have an entire software life cycle every time we commit the code. There are two schools of thought about that. Normally, CI releases to QA.

What Is Build in a Continuous Integration Scenario?

In a CI scenario, a build is more than compiling software. A build is made up of all the operations required to release the software. A build is essentially a process that puts all the code together and verifies that all work fine.

If we consider a typical project, we can see that different people are involved in different areas. The developer creates the feature and, if necessary, changes the script for the table. The database administrator (DBA) puts in place the script and advises the developer when the database is ready. Starting from that, the developer continues the development.

At the end of development, the software is integrated and tested altogether. This can take weeks of work to complete, and in the case of errors, the software may return to the developer for a fix. This costs time and can reduce the quality of the software. This is because, in classic project management professional (PMP), a.k.a. Waterfall, methodologies, quality is a result of three variables:

- Scope
- Time
- Cost

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.

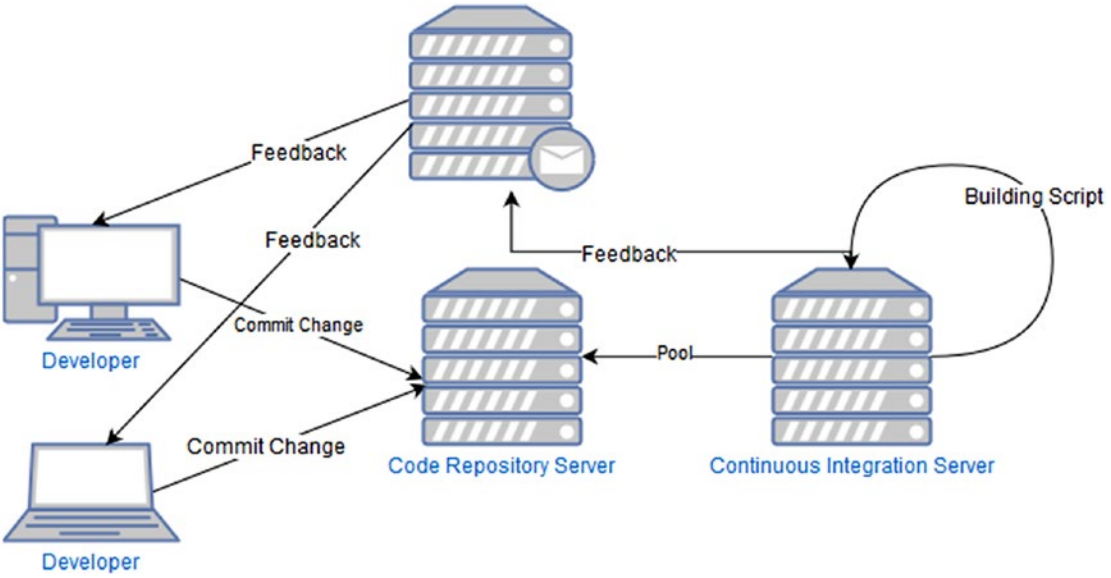

Figure 3-2. *A CI system*

This cycle starts every time the developer commits the code in the repository, which means it can start a hundred times per day.

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).

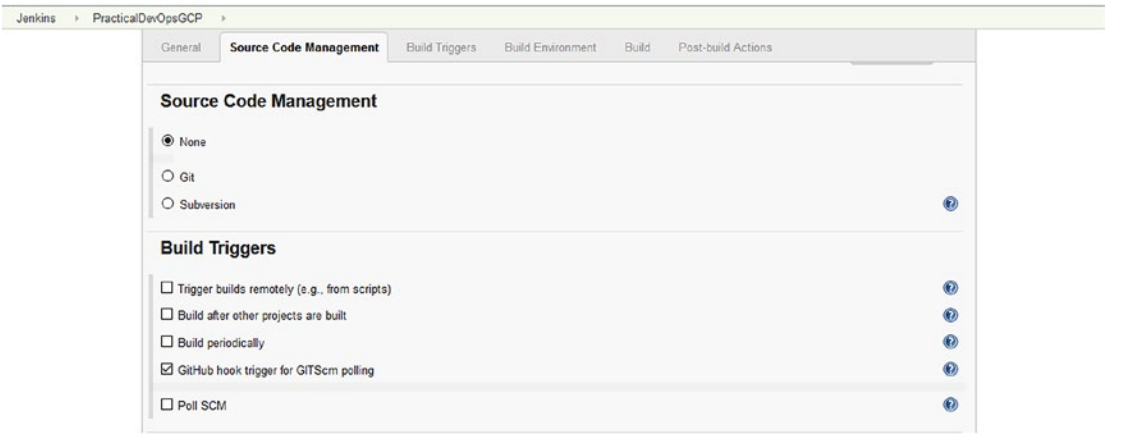

Figure 3-3. *The Jenkins build trigger configuration*

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.

Note When we use a periodic build, we are not really using a continuous integration approach, because the build does not start when the software is committed to the code. This kind of build policy can be good, for example, when we want to have a daily build that can be integrated with a CI policy.

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.

Note We must understand the differences between *continuous delivery* and *continuous deployment*. The concepts are similar, but there is a substantial difference between the two. By continuous delivery, we are referring to a pipeline for creating a build but not necessarily one we intend to release to production. With continuous deployment, we release the build to production every time. This relatively small difference between continuous delivery and deployment makes a big difference to a company's business.

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.

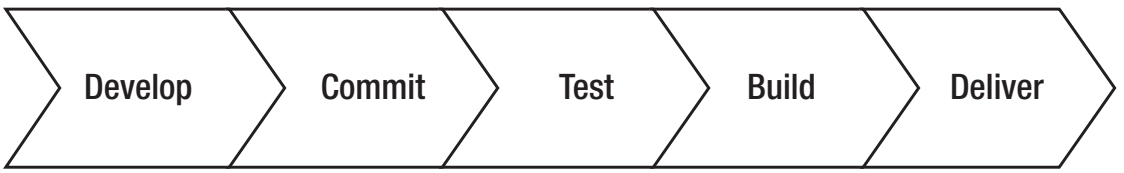

Figure 3-4. *The continuous d chain*

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.

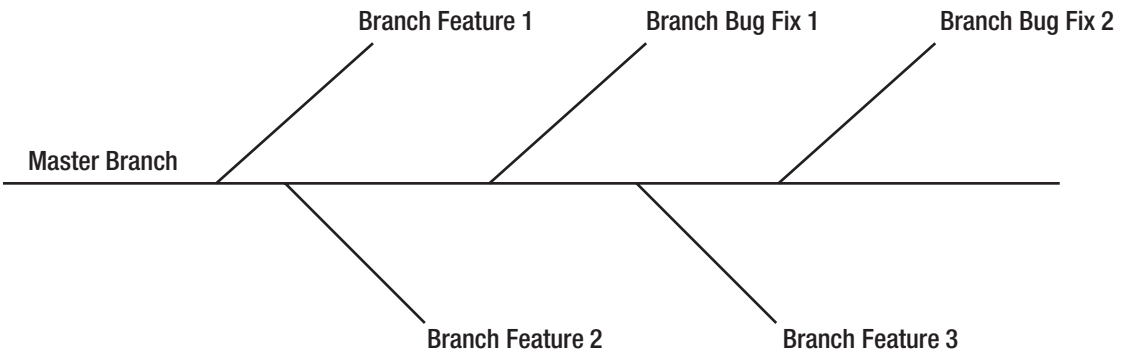

Figure 3-5. *Branching strategies in place*

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.

Table 3-1. *Costs and Benefits of CI and CD*

Practice	Cost/Change	Benefits
Continuous Integration	The developer must write code using the TDD practice. The code must be associated with the unit test.	Because we test every release, we can reduce the number of bugs we release in production.
	We need to put in place a new CI server, which must be used to monitor the repo and start the build on every commit.	A bug is identified soon, which means it can be fixed soon. This accelerates the fix and, of course, saves money
	The developer must integrate the software at least one time per day, which doesn't allow for much difference in integration.	The software is integrated at least one time per day, which means we can have at least one release of the software daily.
	Feedback for every build we make must be in place.	With a CI server in place, we can reduce testing time, because we can execute more tests in parallel and then reduce the time to complete the tests.
		The QA team can reduce the amount of tests needed to be executed to test a single functionality, which means we can spend more time improving the quality of the software testing real-world scenarios.
Continuous Delivery	For CD, we must have strong CI in place, because CD is an extension of CI.	CD reduces the time to deliver software, because delivery occurs every time the process ends.
	We must automate all processes for deploying the software and remove the human interaction.	We can improve the number of releases, potentially to more than one release a day, instead of a big release every six months. This accelerates the feedback provided by customers, which can drive our development.

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.

The most important change we must put in place for a CI system and, later, a CD system is an automatic script for building the software. This script must be used to produce some automatic operation to compile and build the software, and, more important, must be able to start from the command line.

To create the script, we can use software like Maven, Ant, or MSBuild, or we can use the simple command-line scripting in Bash or PowerShell. The language is not important, but we must have something that we can start every time we build under the same conditions that always produces the same result.

The most important change in the CI server is the building block of the CI and CD practice. There are a lot of applications we can use for that. Some are free, like Jenkins, others require a license for professional use, such as Travis CI.

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.
5. 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.

When we have automated this process, we can easily fix any issue with the data simply by calling the process for re-creating the database. This guarantees a set of data that is always correct and removes the chance of having an issue connected to wrong data being in the system. Of course, as with every procedure in CI/CD, this must be adapted to your system. Not all systems can have a complete database release every time. This must be a decision based mostly on the system you are working on.

Note Database migration is not always simple. In some cases, this procedure can be very dangerous. For example, if you work in the financial sector, you don't want to destroy and re-create the database every time you release the software.

With modern development frameworks, such as Rails, .NET, etc., we probably use object-relational mapping (ORM). This frequently includes a procedure known as code first, which means we create the code and, based on that, produce the database. In production, this is not recommended. In this case, adopting continuous database integration can be useful for maintaining ongoing control of 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.

Note Integration testing is an important phase of continuous integration, because it is responsible for testing the entire system and not only the code we develop. When we start the integration test, we essentially bring a different piece of software together and remove any mock library we use in the test. In this phase, we essentially use real data, to test the system and see if it works well with the new software.

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 developer of the code takes the notes and adjusts the code, based on the comments and

asks for another review. When all comments are addressed, the code can be approved and finally merged in the main branch.

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*.

Note Canary servers restrict the number of servers that allow customers to use a feature. The use of canary servers is to reduce potential bugs in production and to obtain real feedback about the release. Because the canary server is the same as that used in the production release, we can use it to gain feedback about the quality of the software. For example, we can see how the software works in a real-time environment and intercept memory leaks and other bugs raised in production.

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.

Note With CD, we don't really have to release the software in production every time. This feature is part of CD, which is important to know.

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).

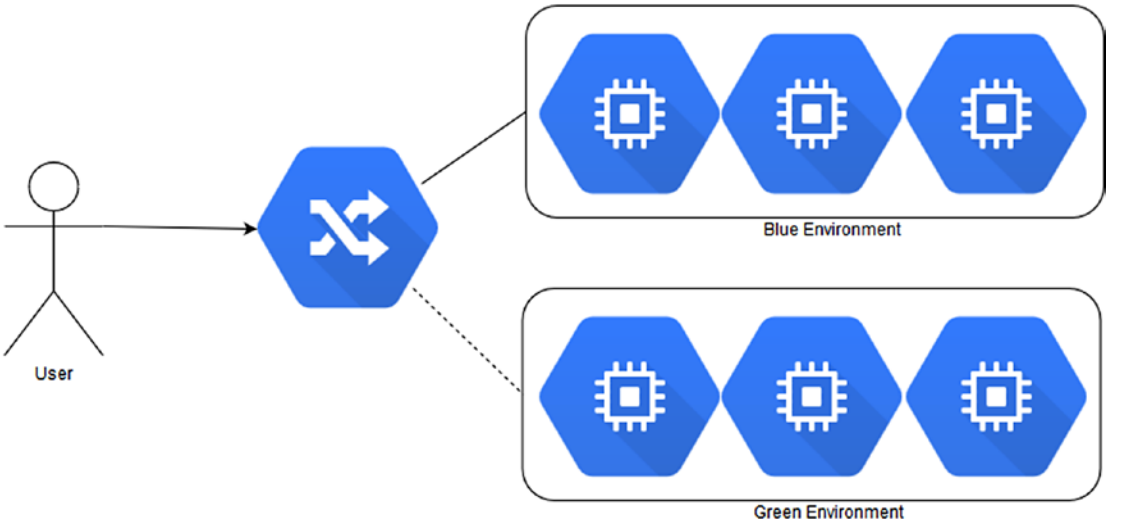

Figure 3-6. *Blue/green deployment*

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% (see Figure 3-7).

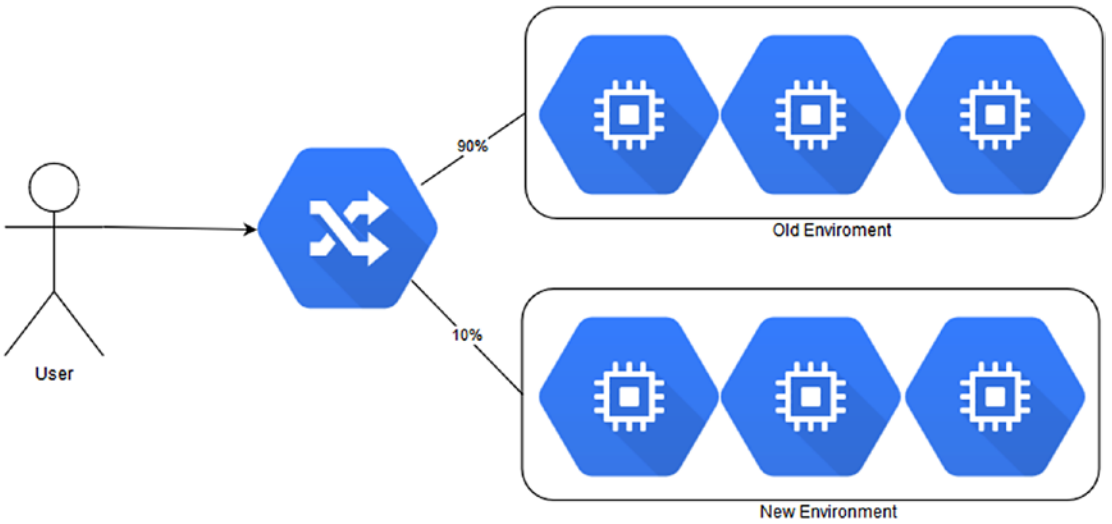

Figure 3-7. *Canary deployment*

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.

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 (Figure 3-8) 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.

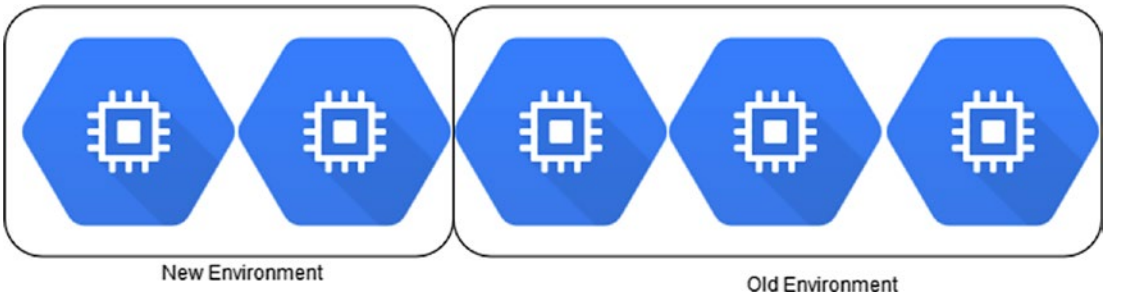

Figure 3-8. *The incremental deployment process*

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.

Conclusion

In this chapter, I discussed CI and CD. Both techniques are at the core of DevOps and are the basis of the cloud deployment.

CI and CD are two closely connected practices. This is because one is the evolution of the other. It is important to spend the time to fully understand these practices to correctly put them in place.

Both practices require a complete change in the way we develop software and how we design the software. We must exert some pressure on the developer to follow the guidelines for CI and CD.

On the other hand, a good system for CI and CD improves the quality of our software and helps to increase release to more than one or two times a year.