Introduction to DevOps

What is DevOps?

The term DevOps represents the combination of Development (Dev) and Operations (Ops). DevOps culture is a set of practices that reduce the barriers between developers, who want to innovate and deliver faster, and operations, who want to guarantee the stability of production systems and the quality of the system changes they make. DevOps culture is also the extension of agile processes (Scrum, XP, and so on), which makes it possible to reduce delivery times and already involves developers and business teams. However, they are often hindered because of the non-inclusion of Ops in the same teams.

Now some key factors to facilitate collaboration and to improve communication between Dev and Ops:

- Frequent application deployments with integration and continuous delivery (called CI/CD).
- Implementation and automation of tests, with a process focused on behavior-driven design (BDD) or test-driven design (TDD).
- Implementation of collecting feedback from users.
- Monitoring applications and infrastructure.

Organizational prerequisites

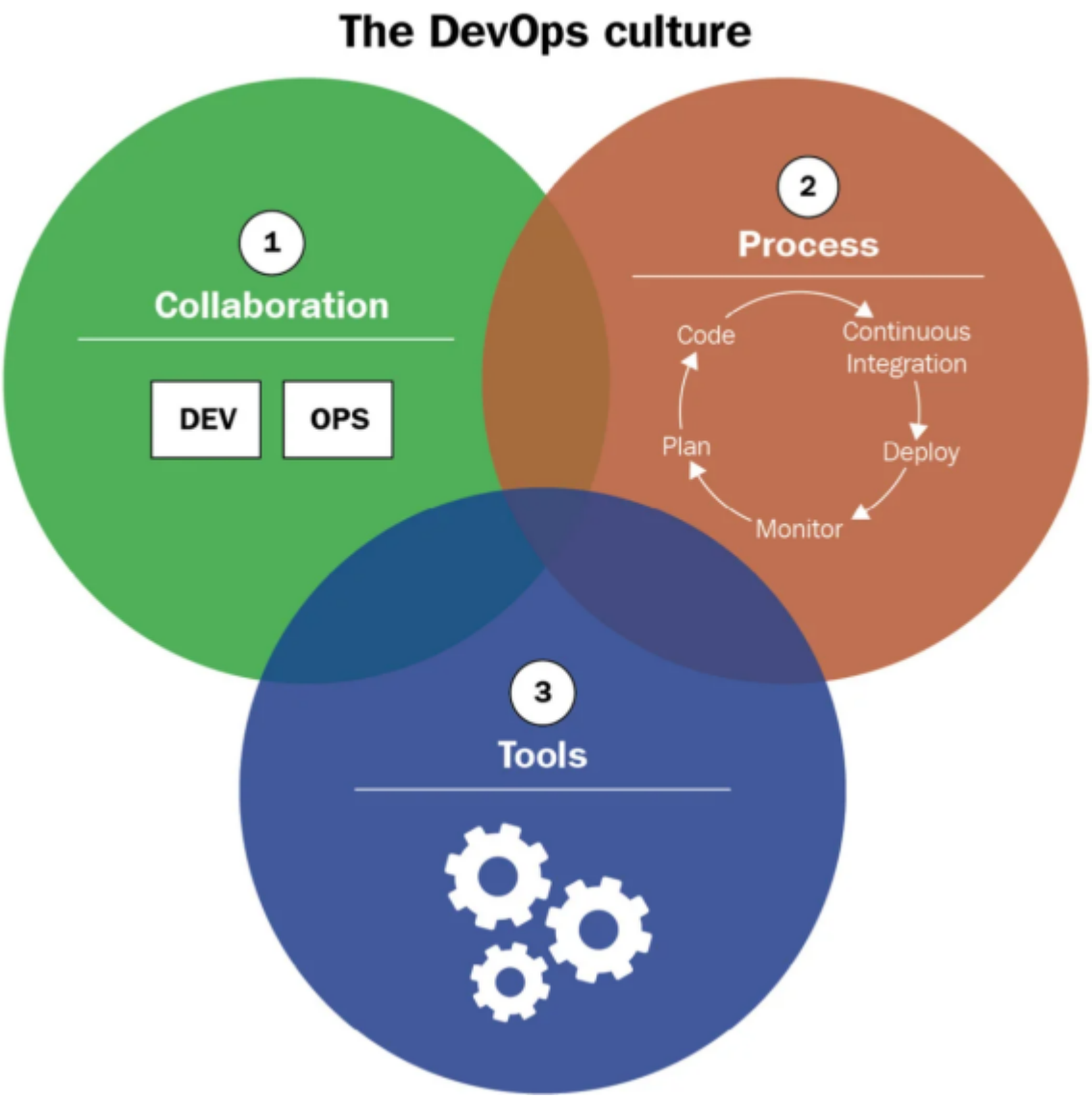
The way your organization works has a high impact on the success of introducing the CD process. It's a bit similar to introducing Scrum. Many organizations would like to use the Agile process, but they don't change their culture. You can't use Scrum in your development team unless the organization's structure has been adjusted for that. For example, you need a product owner, stakeholders, and a management team that understands that no requirement changes are possible during the sprint. Otherwise, even with good intentions, you won't make it. The same applies to the CD process; it requires you to adjust how the organization is structured. Let's have a look at three aspects: the DevOps culture, a client in the process, and business decisions.

DevOps culture

A long time ago, when software was written by individuals or micro teams, there was no clear separation between development, quality assurance, and operations. A person developed the code, tested it, and then put it into production. If anything went wrong, the same person investigated the issue, fixed it, and redeployed it to production. The way the development process is organized changed gradually: systems became larger and development teams grew. Then, engineers started to become specialized in one area. This made perfect sense as specialization caused a boost in

productivity. However, the side effect was the communication overhead. This is especially visible if developers, QAs, and operations are in separate departments in the organization, sit in different buildings, or are outsourced to different countries. This organizational structure is not good for the CD process. We need something better; we need to adopt the DevOps culture.

DevOps culture means, in a sense, going back to the roots. A team is responsible for all three areas, which are shown in the following diagram:



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

- Better collaboration and communication in teams, which has a human and social impact within the company
- Shorter lead times (Lieferzeiten) to production, resulting in better performance and end user satisfaction
- Reduced infrastructure costs with IaC (Infrastructure as Code)
- Significant time saved with iterative cycles that reduce application errors and automation tools that reduce manual tasks, so teams focus more on developing new functionalities with added business value.

Implementing CI/CD and continuous deployment

Key practices of DevOps are:

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

Let's look in the following chapters at each of these practices in detail.

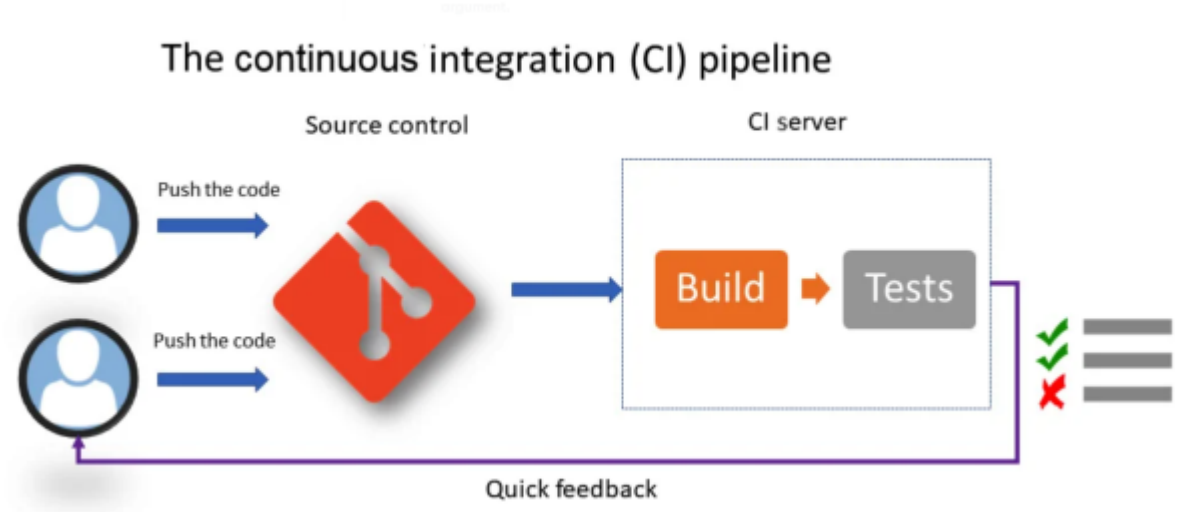
Continuous integration (CI)

Continuous integration is an automatic process that allows you to check the completeness of an application's code every time a team member makes a change. This verification must be done as quickly as possible. To set up CI, it is necessary to have a Source Code Manager (SCM) that will centralize the code of all members. This code manager can be of any type (i.e. Git or SVN). It's also important to have an automatic build manager (CI server) that supports continuous integration i.e. Jenkins, GitLab CI, TeamCity, GitHub Actions.

A CI-server will retrieve the code and then do the following:

1. Build the application package (compilation, file transformation etc.)
2. Perform unit tests (with code coverage)

This diagram below shows the cyclical steps of continuous integration. This includes the code being pushed into the SCM by the team members and the build and test being executed by the CI server. The purpose of this process is to provide rapid feedback to members.



Continuous delivery (CD)

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

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

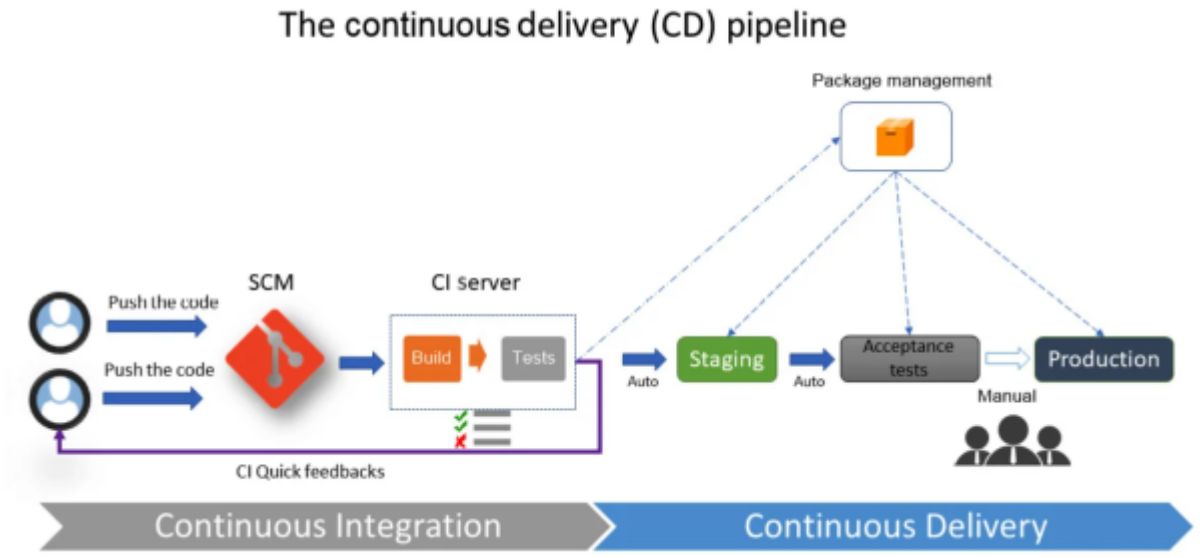
Unlike CI, CD aims to test the entire application with all of its dependencies (i.e. a microservice application composed of several services and APIs). It is important that the package that's generated during CI is the same one that will be installed on all environments, and this should be the case until production. There may be configuration file transformations that differ depending on the environment, but the application code (binaries, DLL, Docker images, and JAR) must remain unchanged.

The tools that are set up for CI/CD are often a

- `package manager`: This constitutes the storage space of the packages generated by CI and recovered by CD. These managers must support feeds, versioning, and different types of packages. There are several on the market (Nexus, ProGet, Artifactory).
- `configuration manager`: Most CD tools include a configuration mechanism with a system of variables.

In CD, deploying the application in each staging environment can be triggered …

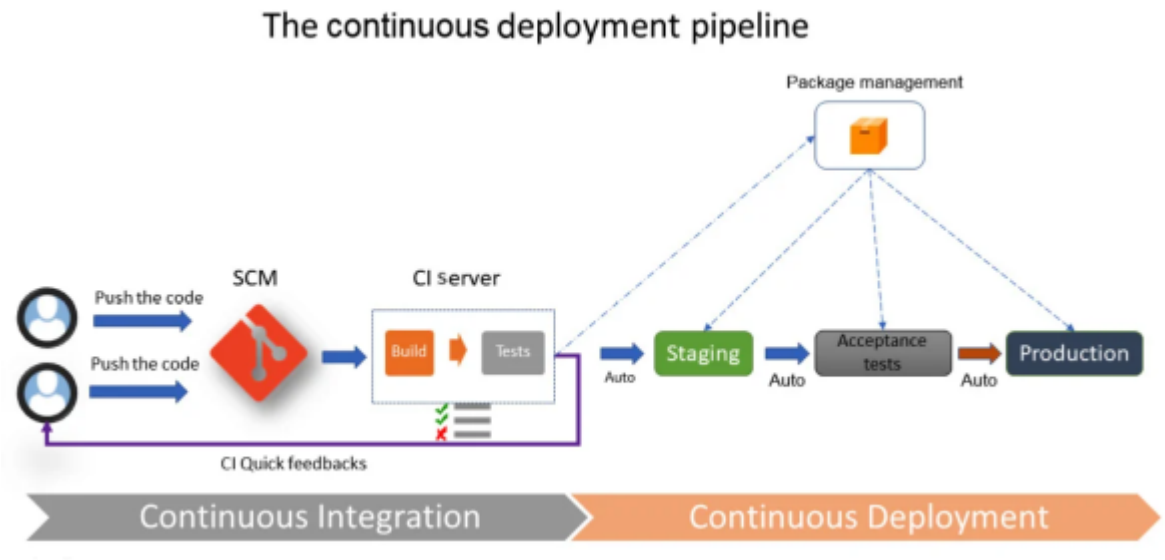
- `automatically`, following a successful execution in a previous environment.
- `manually`, for sensitive environments such as the production environment, following manual approval by the person responsible for validating the proper functionality of the application in an environment.



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

Continuous deployment

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



This practice is rarely implemented in enterprises because it requires a variety of tests (unit, functional, integration, performance, and so on) to be covered for the application. Successfully executing these tests is sufficient to validate the proper functionality of the application regarding all of these dependencies. However, it also allows you to automatically deploy to a production environment without any approval action required.

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