Introduction to CI/CD

In this lesson you will learn about the need for automation in your microservices development lifecycle and why Continuous Integration and Continuous Deployment are key practices that you must embrace to ensure the fast delivery of high-quality services to your customers.

The need for automation

Let's review the process that John, one of your engineers, is currently using to get one of his microservices built and deployed from his local development environment to Production.

As usual, John stars by writing the code for the next feature or bug fix. He will compile his code likely multiple times until arriving to a version that meets the requirements.

John will then run a series of either manual or automated tests to make sure the changes behave as expected and there are no regressions to any of the microservice functionalities.

After this, he will build a Docker image, tagging it with a new version, and then he will publish it to the container registry.

Finally, John will update the version of the docker image in the Kubernetes deployment file and he will deploy new pods with updated microservice containers.

This immediately shows the first challenge for John to get things from his dev box to production. For every change he needs to perform multiple tasks beyond just writing, compiling, and testing his code, which heavily impacts his productivity.

This is already bad. But then what happens if, for any reason, John forgets to run a few of his verification tests before a new deployment? He will likely end up with a bug all the way into the Production environment, which could have serious consequences for the system and his customers.

And, of course, to fix the bug John would have to either undo his latest code changes or write some new code, which has to go through the entire sequence once again.

So, having to work on these dev ops tasks manually makes it quite easy for bugs to make their way to Production.

Also consider that, eventually, John will push his code to a remote repository, like GitHub, from where other engineers, like Jane and Peter here, will pull it into their local dev environments.

Eventually, they will also add their own code to implement new features or fix some bugs.

Let's say Peter is working on a bug fix that involves a simple one-line change. Peter is in a rush to go home since it's late Friday already, and his machine is taking too much time to get anything built, so he just pushes his code to the remote repo without compiling.

Sadly, it turns to be that his change has a syntax error that broke the code, and now that code is on GitHub. Early Monday, Jane and John will pull the latest code, hope to get a quick start on the week, just to find out that nothing even builds.

As you can see, the lack of automation can certainly lead to broken code that will impact the productivity of all team members. Plus, it blocks the team's ability to get anything shipped.

Finally, since the whole build, test and deploy process is done manually, in this case by John, he will have to ask the team for their status to know if the latest code has been verified properly before starting the deployment tasks. John can't possibly know how to verify all the changes introduced by each team member.

So, without some sort of centralized automation, none of the team members can be completely confident about releasing new bits to Prod, which significantly impacts the team's agility.

There are even more issues with this approach, like making sure a machine with the right prerequisites is always available to produce and deploy a build, or what happens if John is just not available to deploy an urgent fix, since he is the only one that knows how to deploy things.

Fortunately, CI and CD can greatly help to address these, and many other related developer challenges.

What is Continuous Integration (CI)?

The first part of CI/CD stands for Continuous Integration, and this is what it stands for:

Continuous Integration, or CI, is a software engineering practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run.

The key goals of continuous integration are to find and address bugs quicker, improve software quality, and reduce the time it takes to validate and release new software updates.

We'll see how exactly CI integrates into the software development process in a minute, but let's first also define the second part of CI/CD.

What is Continuous Delivery (CD)?

Continuous Deployment, or CD, is a software engineering practice in which code changes are delivered frequently through automated deployments.

CD expands upon CI by deploying all code changes to a testing environment and/or a production environment after the build stage.

Let's see how the CI and CD practices modify the manual build, test and deployment process that John and his team have been using so far.

Enter CI/CD

Just as before, John goes through the usual tasks of writing code, compiling and running tests, in order to implement the latest feature or bug fix. These tasks are always expected from a software developer, so no changes there.

However, after verifying his code locally, there's nothing else he has to do other than committing and pushing his code to the remote repository.

From there, a continuous integration engine, like GitHub Actions, can take care of making sure the code compiles and passes the latest set of automated tests. This is the continuous integration phase, and it kicks off automatically as soon as the latest code has been pushed to the remote repository, with no manual intervention from the software engineer.

Once the latest bits have been verified, the next phase automatically kicks off. This is where a new version of the microservice docker image is created and published to the container registry and is followed by a deployment to the Production environment in the form of Kubernetes pods.

This phase is known as Continuous Delivery, or CD, and when combined with CI, they form what is known as the CI/CD pipeline, which presents multiple benefits.

As you can tell already, the developer productivity is significantly improved since developers can focus on writing quality code as opposed to dealing with Dev Ops tasks.

Also, CD will make sure that the latest verified version of your microservice is always deployed and ready to be used by your customers.

Now, what happens if John, once again, forgets to run some of his automated tests locally? Well, thanks to the CI/CD pipeline, the bug can be quickly caught as a regression during the automated testing phase.

And, when this happens, the pipeline will stop any further phases from happening and can send notifications to John and anyone else interested in knowing about the health of the build.

So, a CI/CD pipeline is a great way to catch regressions early in the software development process, which can save the team a lot of effort and dollars.

Let's also bring in Jane and Peter, who once again have pulled the latest changes into their boxes and are working on new features. Again, Peter is in a rush to leave the office, and he would love to just push his code to the remote repository to be done with it.

However, and fortunately, pull requests have been enabled as a mandatory requirement in the repository, so the best he can do is send a pull request with his uncompiled code and go home for the weekend.

The CI/CD engine will now try to compile the code in Peter's pull request and, as soon as it detects that the code is broken, it stops the pipeline and then notifies him of the failure.

Thanks to this, broken code can't propagate across dev environments, saving a lot of time to all his teammates. And, even if the code somehow managed to get merged into the remote repo, the pipeline would detect the failure and prevent any faulty bits to reach Production.

With a CI/CD pipeline there's also no need for coordination between team members to decide when and who should oversee deployments. The pipeline will take care of it and can notify anybody interested any time a new build hits Production.

Also, notice that when using a CI/CD engine like GitHub Actions, there is no longer a need for developer boxes to get the latest bits built and deployed. The engine will provide the necessary compute power and a way for you to define the environment prerequisites to build, test and deploy your microservices.

Finally, the knowledge regarding how to build, test and deploy things is no longer concentrated in few team members, since a good CI/CD engine will allow the entire process to become part of the code itself, making it easy to understand and modify if necessary.

In the next lesson you will learn how to use GitHub Actions to enable continuous integration, or CI, for one of your microservices.