Continuous Delivery and Deployment – Part 1

This lesson discusses continuous delivery and deployment.

We'll cover the following

What is continuous delivery?

Continuous delivery is a software delivery approach that enables engineering teams to deliver patches, updates, and new functionalities to the end-users over and over continually in short periods of time in a reliable and efficient way.

With the adoption of this approach, developers can incrementally push even the smallest of the code changes/updates to production. There is no need to hold the deployment until there is a big chunk of updates, that is accumulated to be pushed to production.

With this approach, software can be released multiple times a day, weekly, or whenever it fits best for the business.

Why is this quick push to production a big deal? Why not accumulate small changes and just push one major change?

Why adopt the continuous delivery approach?

Here are a few upsides of adopting the continuous delivery approach:

1. Easy troubleshooting and rollbacks

When small incremental changes are pushed to production, it's easy to

When small incremental changes are pushed to production, it's easy to troubleshoot issues if they arise. It's also easy to do regression testing and so on

because everyone is aware of the small code's impact on the system. Things are comparatively easy to comprehend.

When the number of code changes pushed at one time becomes greater, tracking things gets a little tricky.

Also, small incremental changes can be easily rolled-backed without breaking anything. Imagine pushing multiple patches, clubbed together, to production at one time. Say we push five small patches rolled into one to production together and, out of those five patches, one causes an issue. Now, because of that one patch, all five patches have to be rolled back. We might also have a hard time figuring out which patch went rogue.

Therefore, pushing incremental changes is a cleaner approach.

2. Increase in team's productivity

Since there are no planned releases, developers can have their code pushed to production as soon as they commit it to the remote repository. The developer pushes their code after performing a thorough testing and code quality check on their local machine, and in a dev environment if required.

Once they are happy, they can push the code to the remote repository and have their update go through an automated process of builds, different tests, and quality checks and directly to the production through a deployment pipeline.

This continual release approach cuts down a lot of time required to push the software to production, increasing the team's productivity. There will be more on this later.

Life before continuous delivery

Back in the day, I worked on a big telecom project. The business came up with all sorts of fancy new voice call and data tariff plans for both their paid and prepaid customers on an on-going basis. Boy, it was intense making changes to the code and writing new features so frequently. Also, we did not have any sort of continuous delivery deployment model back then. The entire deployment process was manual.

Pushing the code to production was really something. It was an effort.

This is roughly how it was done: After the code was written, it was pushed to multiple staging environments. The builds, tests, quality checks everything was run manually.

After the builds, tests, and other quality checks were successful, we had a manual testing team test all the changes, including performing the regression testing. They had to test if the new changes broke any of the existing functionalities.

Also, there were no automated browser tests, no selenium, etc. Everything had to be manually tested from scratch, and only when the testing team gave the green light, the code was released to production.

We generally choose one day on the weekend, usually Saturdays, to push the code to production. Also, all the development across different teams was put on hold during the release.

You can imagine the amount of time a small code change took to move to production. I mean this wasn't even feasible. Every small change had to be accumulated into one for a major release. Writing the code took 50 to 55% of the time. The rest of the time was dedicated to testing and deployment. This was a major time sink, and it impeded productivity. If things were automated back then, it would have doubled the feature release velocity of the team.

Although I agree, not everything can be automated, and there are scenarios where we cannot do without manual testing. Still, the *continuous delivery* and the *continuous deployment* approach would have saved us a lot of time.

What is continuous deployment?

Continuous deployment is a term that has a meaning similar to *continuous delivery*. The only difference is that in continuous deployment the entire software release process is automated. There is zero manual intervention.

Automating everything is tricky because we often have to place manual checks in between the pipeline to ensure things work as expected. This becomes more important when the software being developed is for an airline company, a medical life support system, and so on. We cannot just let the automation take over and decide everything for itself.

Automated tests never cover the product comprehensively. Therefore, we have to look out for the corner cases manually. We need humans for that.

Businesses often leverage both the *continuous delivery* and *continuous deployment* approach to build their deployment pipelines. *Test-driven development* and *containerization* help big time in the implementation of both the approaches, facilitating a quicker move to production.

Test-driven development ensures a good code coverage and thorough testing of code behavior. *Containerization* facilitates managed deployments.

Let’s continue this discussion in the next lesson.