**Continuous Integration & Continuous Delivery**

Even junior developers need to gain familiarity with continuous integration and delivery because these concepts sit at the heart of modern development. These processes sit at the heart because they bind the work of a team of developers together. Many new developers coming from university (or being self-taught) have not really had the experience of working on a large codebase as part of a team. As a result, even if they have used these platforms, they may not really understand the point.

Let’s look at how these concepts work with the growth of a team. This is a fictional company with fictional events, but it’s meant to illustrate how a (very naive) development team could grow and strain and learn to use CI and CD the proper way.

**A single developer**



Most small projects are a single developer working on their own code. Code organization is flexible or non-existent. There is little need, as the whole codebase sits inside the sole developer’s memory.

The developer is really plays all the roles that are normally part of a larger team: they decide what they want to build (business owner/stakeholder), they prioritize when and how long to spend building each feature (project manager), and once done they hopefully test their work (QA).

The application may or may not have users other than the developer. If the app is used, then each time the developer makes a change, they need to deliver that update to their Users. In the past (and still today for single person teams), this might just involve using SFTP to send a jar file or upload files to a server and restarting the app.

For now, we’ll assume that the dev is working on their own with no end users and that they are working on a web application.

**Two developers**



Now the two developers must communicate — who works on which features, and how they will work. With even this minuscule team, there becomes a need for clear APIs and regular updates to work. It’s no longer enough to test your own work — the moment your code depends on the other developer, the code must be integrated and tested as a whole. Negotiating when these combinations and tests would happen would eat up a lot of developer time.

Wouldn’t it be great if after every change this happened automatically?

In order to do so, a few things must be agreed:

1. **When to trigger code integration tests**
2. **How to test that the integration has been successful**
3. **How to communicate the results to the team.**

All continuous integration platforms have their default way of answering these questions. Most allow quite a bit of configuration and setup (sometimes an overwhelming amount).

Typically, **CI platforms trigger integration tests on a merge to the master branch in git**, or the equivalent if you are using one of the other code management tools. On merge, the **CI system will execute a set of tests and declare that the code either integrated successfully or not**.

**For compiled languages, there is a single default test: does the code compile? If not, then the build is broken and integration has failed**. **For scripting languages like Java Python or JavaScript, there are no such default tests so developers must create their own.**

**Either way, most CI systems show a history of integration attempts, success/failure, and usually have a way for the system to send an email to the team if things have broken**.

**The two developers work on their code and write simple tests for each major piece of work to show that things are working at a basic level. They amend their script to run these tests and fail the integration if any of the tests break. For now, all is well and work continues**.

**Two developers and a few users**

Our team of developers finally thinks they have something to show to real customers. It’s time to put the software out for the world to see.

Now there is a new issue to solve, different from integration. How do the developers get their code to the users? Again, they could use SFTP or something to move code to a server, but this quickly gets repetitive and boring.

This is where **continuous delivery starts to come into play**. The two developers agree that every successful build, they will have a script in the CI server zip the files and with a button they can push the latest build to the server and restart the application.

A bit of hackery later, and it works. Each time code is merged to master, the integration server creates a new code build, and one of the devs hits the button to push the code into production for their users to enjoy.

This works for about 2 weeks when suddenly a massive problem happens. Somehow, a major (but subtle) bug has been introduced and the few users they have are angry.

Now there are a few issues: when did the bug get introduced? How can they roll back safely? More importantly: how to win back the trust of their users?

**Two devs, QA, and users**



The two developers manage to hack a quick patch in and get things running again, but now they are scared. Some users were really bothered by the earlier flaw and in retrospect, it could have been caught by playing with the app sufficiently.

The developers confer and decide to hire someone to do initial testing and feedback and to write larger tests. As part of their complaints, the users not only mentioned that something was broken, but that other parts of the app were “just weird”. It’s decided that another pair of eyes would be really helpful.

But there’s a catch: until now almost every build was getting sent to production. The delivery to users was continual, sure, but the consistency of what was delivered was not. While the new QA engineer can try and find issues, they should probably do that before everyone else sees the build.

The three team members discuss and decide to set up a second server that they creatively name “The QA server” and it’s that server that gets the latest build. If the new QA engineer is happy, they are allowed to hit a button in the CI/CD software that will deploy to production.

The new QA engineer gets to work, writing integration tests, automated QA tests in selenium, etc. All of these tests have to run somewhere, so they get added as extra steps in the build pipeline.

**The team grows**

The app continues to be a success. The two devs are beyond overwhelmed. They hire three new developers, a designer, and a project manager who also responds to user email in order to provide some basic customer support that feeds into feature requests and bug fixes.

The QA engineer gets frustrated. With so many people doing work, things are far less stable. A lot of builds are duds — too many people working too fast and the merge doesn’t go quite right, etc. The developers reach out regularly and say “Don’t touch that! we know it’s broken!”. The QA leader suggests that the team create yet another server — called development — which will be up to them to control. Whenever they are satisfied that all the pieces work, then they can send a complete build to the QA server. There is no point in running a bunch of QA scripts against an app that’s known to be broken.

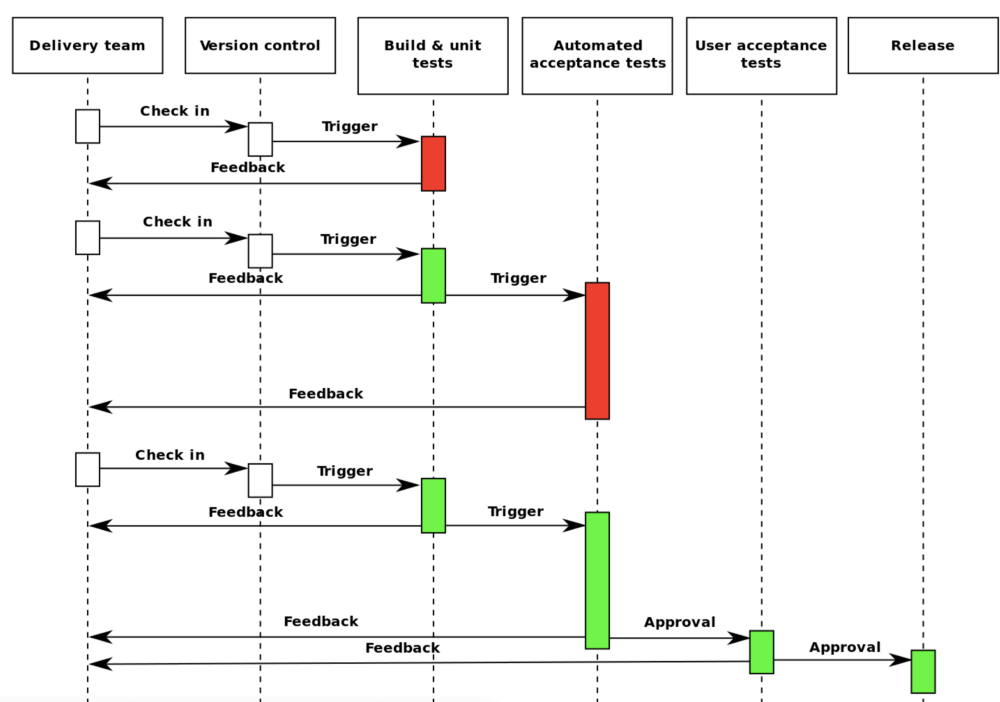
**The first big user**

A large enterprise becomes interested in this awesome web app our micro-team has been developing. They reach out and want some custom features. They are happy to pay but want to be able to approve functionality before it gets rolled out to their staff.

The team members discuss and decide that they are happy to build out these features but they don’t want to support separate apps. Anything the enterprise user wants will have to eventually get rolled out to public users. Miraculously, the enterprise agrees, provided they get a discount on pricing. Everyone is happy and life goes on.

However, to support this use case the team decides to create yet another server, called “Acceptance”. This server will be for these key users to try new features and give a thumbs up or thumbs down.

Now we have something that looks very much like this:



Now we have a full continuous integration and continuous delivery pipeline. The software can be developed quickly, tested quickly, and approved quickly. The team finds they spend very little time on anything that doesn't deliver value.

**Conclusion**

The evolution of continuous integration and continuous delivery often looks a lot like our story. Delivering quality software quickly and efficiently isn’t easy at the best of times, so it is important that developers have systems that support them.

I hope new developers gain some insight from this story, and consider how it relates to their current organization and their pain points. Are there issues with quality? Delivery? Debugging? User happiness? It’s all connected to the software lifecycle. Fixing the lifecycle and the tools that power it can make a huge impact.