Editor: **Diomidis Spinellis**
Athens University of Economics and Business, dds@aueb.gr

Continuous Integration and Its Tools

Mathias Meyer

Post your comments online by visiting the column's blog: www.spinellis.gr/tools

CONTINUOUS INTEGRATION HAS been around for quite some time now. The idea was originally conceived as part of the practices collectively known as extreme programming. At its very core, continuous integration is a set of principles that apply to the daily workflow of development teams. Let's walk through the basics.

All code must be kept in a repository. Git is a common tool, but you'll also find Subversion, Perforce, Mercurial, and an abundance of other choices in active use today. Surprisingly, you'll still find places where using a code repository and versioning the source code aren't yet part of common practice. In a continuous integration life cycle, when someone checks his or her revised code into the repository, an automated system picks up the change, checks out the code, and runs a set of commands to verify that the change is good and didn't break anything. This tool is meant to be the unbiased judge of whether a change works or not, thereby preventing the "it works on my machine" syndrome before the code hits production.

For all this to work, the build must be automated, and thanks to modern Web frameworks and build tools, this is an easy-to-achieve task. A build ideally involves more than just compiling—it should also include a thorough test suite to help verify that the code still works with every change. The objective is to increase test coverage with every new change, but also to keep the test suite fast. If tests run longer than 10 minutes, developer productivity drops, slowing down the process of shipping new features or bug fixes to the customer.

Commit Daily, Commit Often

A core practice of continuous integration is that all developers commit to the mainline branch daily. With modern, distributed version control systems such as Git, the temptation is high to work in a local branch, maybe pushing your work to a remote repository and eventually merging your changes. What if, instead, your developers made all their changes on the mainline branch? Some even go as far as discarding work that's not committed by the end of the day, to make sure the habit really sticks.

When your team makes changes in smaller increments and integrates them into the mainline regularly, the benefits go beyond just the development process. Smaller changes, shipped to production quickly, are a lot easier to debug when something breaks. This is one of the key values of continuous integration, and it's where the name comes from. Rather than living in branches for long chunks of time, changes are continuously integrated.

Build Early, Build Often

How do you verify whether someone's changes broke something in the code or whether the changes work in the larger

0740-7459/14/$31.00 © 2014 IEEE

context of the entire codebase? Beyond version control, a continuous integration server is one of the more important tools a development team can put to good use. Its sole purpose is to check the code repository for changes, check out the code if it spots any, and run a list of commands to trigger the build.

The definition of a build varies widely depending on the language and framework used. With compiled languages such as Java, C++, Erlang, and the like, a build is a compilation step with an additional run of a test suite. For dynamic languages such as Ruby, Python, and JavaScript, a test suite is commonly the only thing that's run as part of the build.

A continuous integration server is unbiased. Its tasks boil down to telling the team whether the most recent changes still pass the stages it's configured to run. An abundance of tools are available, from self-hosted systems such as Jenkins, TeamCity, and Bamboo to hosted products like CloudBees and Travis CI. They're all very easy to set up, so there's no excuse not to use a continuous integration server for your projects.

The last step of a fully automated build is the ability to deploy to production, which requires an automated deployment process that every developer should be able to run just like the continuous integration server. With an automated build in place, everyone can deploy to staging or production, anytime. What if this could even become part of your build process? Imagine shipping all new code directly to the staging system for instant testing!

Keep It Green

With everyone committing to the mainline, a green build is an important responsibility. A green build is the result of a code change passing all the steps involved in the build successfully. When developers break the build, they're blocking everyone else on the team from committing their changes with confidence. When the build is broken and other developers keep making changes and committing them, it's a lot harder to figure out whether or not they failed the build, making debugging the failed build harder with every new commit.

At Toyota, everyone on the factory floor can halt the production line if something breaks or holds them up. In your development team, a broken build should spur a similar habit. Getting it back to green should be the highest priority. A green build is your insurance policy that changes can be deployed to production at anytime.

A Matter of Culture

Beyond having good tooling and a fast test suite in place, continuous integration is really about fostering responsibility and providing customer value. The server to run the build is but a small part of it. The practices that continuous integration includes have a much bigger picture in mind: a responsible team that ensures new features can be shipped to production and the customer quickly, and that makes sure all members can get their work done and check in new code.

When new changes are continuously merged into the mainline branch, there's a deeper responsibility for everyone on the team to make sure that those changes not only pass the build but that they also have minimal impact on the

production environment. Rather than build new and bigger features on long-lived feature branches, developers start thinking about how they can build them in ways that let them merge their work into the mainline branch regularly. Feature flips are a popular way of reducing the impact of new features and providing ample means to try them out before shipping to all customers. They have the added benefit that when something breaks in production, features can easily be switched off to reduce the load on the system while the issue is investigated.

But how do you notice that something is broken? How can you know that a change doesn't have a negative impact on the production environment or on your customers? Good monitoring is essential: logging, metrics, alerting, and exception tracking will help you ship changes with confidence. Thanks to an abundance of hosted services, it's just as easy to get started with monitoring as it is with continuous integration servers.

Feature flips and monitoring aren't practices originally considered with continuous integration, but they're invaluable additions to the process of shipping early and often. With a responsible team focused on shipping and increasing customer value, continuous integration can be a catalyst for long-lasting change, just by adding a few simple practices to your team's workflow. When you add these habits together along with feature flips, fully automated and fast deploys, and monitoring, you're ready to ship value to your customers anytime.

MATHIAS MEYER works at Travis CI, a hosted continuous integration and deployment platform for open source and private repositories. Contact him at meyer@paperplanes.de or via Twitter (@roidrage).

See www.computer.org/software-multimedia for multimedia content related to this article.