Super-powered continuous delivery with Git

Now that Git has solved the pain of merging, branching workflows are far more attractive.

BY SARAH GOFF-DUPONT

We've all heard the saying "beware of code written by just one person," and we know the benefits of making software as a team: you get different ways of thinking, different backgrounds and experiences... and when you bring those differences to bear on whatever problem you're trying to solve, you end up with better software. It's more maintainable, higher quality, and ultimately serves the user better.

But here's the other thing we know: developing as a team can be messy!

You're trying to understand which pieces of code everyone is working on, trying to make sure changes don't conflict, trying to find defects before your customers do, and trying to keep everyone connected with the project up to date on how things are progressing. As it turns out, each of those problems is addressed by either Git branching or continuous delivery.

I hope to convince you that, by combining the two (maybe throw a bit of kickass tooling into the mix just for fun), you've got a super-powered recipe for success.

The power of branch-based workflows

Truth be told, it's not that Git in and of itself is so perfect for continuous delivery. It's that branching workflows are great for CD, and Git is great for branching workflows. Aside from being CD's BFF, a branch-and-merge workflow lets you tackle thorny bugs, try out new technologies, or just start coding a new feature from scratch *without the risk that your changes will prevent your teammates from forging ahead with their own tasks.*

Clearly, Subversion and other traditional version control systems let you branch, too. But let's side-step for a moment and meet branching's evil twin: the merge.

Traditional version control systems like Subversion are simply not that great at tracking versions of files that live on different branches, and when it comes time to merge, Subversion has to stop and ask for directions a lot. (You know... that little pop-up asking "Do you want this line or that line in the merged version?") The fact that so much human interaction is required during merges drives teams to institute code freezes so whomever is performing the merge doesn't get disrupted by new changes coming in on one of the branches. And code freezes are expensive: it's pretty un-productive time.

Git, on the other hand, is really good at tracking changes to different versions of files that live on different branches, and it always knows what the common ancestor of that file looked like. So it basically has a built-in GPS that lets it navigate merges without having to stop and ask you for directions all the time.

With Git, you're free to exploit the power of branching in a way that would be impractical with Subversion. The overhead involved with branching and merging is so trivial that branches which live only for a day or two become not only feasible, but beneficial.

Ok, fine. But what exactly makes branching so powerful for continuous delivery, anyway?

Branches keep main clean and releasable

We've established that short-lived branches provide a great way for developers to collaborate on a project or feature without stepping on each other's toes. But more importantly for CD, isolating all the work in progress on development branches keeps main and any stable version branches in a clean state so you can ship at will.

This means running your full battery of automated tests against dev branches so developers have strong signals about the quality of their code, and can make confident decisions about when their changes are ready to be merged up. (If you're not rocking the automated tests yet, check out this post from RebelLabs for a playful talking-to and recipes for writing your first unit tests.)

It also means using Git's pull requests as a form of code review so your entire team is confident in the code's maintainability and interoperability with other areas of the code base. Yes, this is more up-front work than traditional delivery models call for. And yes, it's worth it.

Successful continuous delivery hinges on keeping your release branches squeaky clean.

By way of example, all the development teams at Atlassian have an agreement that nothing is ever committed directly to main or the stable branches. Everyone does their work on branches. In fact, we're so bullish on branching that we've taken to creating a separate branch for each Jira issue we tackle – more on that in a bit.

Anyway, this means that people can break as many tests and do as much damage on their branches as they want! Main remains in a state where we can release from it, and where you can make new branches off it without inheriting a bunch of broken code. That's a win for CD and general developer productivity (not to mention morale).

Branches help you accept contributions from outside the team

Git's facility for branching – and especially for forking entire repos – makes it easy to take contributions from people outside the immediate team: contractors, developers from partner companies, devs from other business units, etc. You don't have to lose sleep worrying about people who are unfamiliar with your code base making changes on critical branches willy-nilly and thwarting your ability to ship new code.

Here again, rigorous automated testing on their branches or forked repos is the key to collaboration happiness. You'll want to review their build and test results before approving any merges into your main code line.

> **Pro tip:** Repository managers like Bitbucket let you use Git hooks to enforce quality standards. For example, you can set a rule that all branch builds must be passing before a pull request can be accepted and merged in.

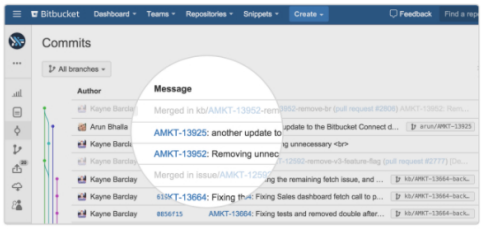
Branching done right = project tracking clarity

Now trending: create a development branch for each story or bug-fix or task (ie, each JIRA issue) you implement. We adopted this branch-per-issue model at Atlassian a couple years ago, and haven't looked back! It's become popular with our customers, too.

Creating a branch for each issue makes it easy to hand-pick which changes to ship out to production or bundle into a release. Since you're not dog-piling changes onto main, you get to select what comes into main – and when. You can ship an epic's MVP plus one nice-to-have, rather than wait until all the nice-to-haves are fully baked. Or ship a single bug fix, and do it within the framework of a regular ol' release. Even if the fix is urgent, you won't have to deal with the 3-ring circus of backing out other changes that aren't ready to ship yet just to get that one change out the door.

And that ease of shipping a single code change is the essence of continuous delivery.

Not only does this approach keep un-proven code off main, when you include the relevant Jira issue key and developer's name or initials in the branch's name, it's crystal clear what the state of development is for each issue in flight.



Notice the naming convention used in the picture above: it's the unique key for the JIRA issue being implemented, plus a short, human-readable description of what that issue is all about. So as a release manager or other stakeholder, you could look at the repo shown above and know at a glance that the user story tracked by AMKT-13952 is ready for release because you can see that it's been merged to main. That's traceability without all the manual effort – boom.

So how does the Git + continuous delivery workflow work?

Glad you asked! I'll go through it quickly here at a high level since other articles on this site dive deeper into each phase.

- **Create a branch for the issue you're about to work on.** Include the JIRA issue key in the branch's name so it's clear what the branch is for. And if you're using other Atlassian tools like Bitbucket or Bamboo, they'll pick up on that issue key and create links between the issue, your branch, all your commits, your builds, your pull requests, and your deploys related to that issue. In other words, issue keys are magic.
- **Make your changes on the branch.** You're off in your own little world here, so go nuts. Try new things. Break stuff. It doesn't matter, because you'll also...
- **Put your branch under CI.** (Bamboo does this for you automatically, btw.) It's up to you and your team to decide whether to run specialized tests like load tests or end-to-end UI-based tests here, and whether to automatically trigger a test run each time you push changes to your branch. Atlassian's teams generally run unit- and integration-level tests on development branches, and let the developer choose how often to run them in order to conserve build resources and make sure the queue isn't unnecessarily clogged.
- **Update your branch with the latest from main frequently.** You can use rebase or branch-commit to accomplish this. Totally up to you. (But remember not to rebase a branch that you share with other devs – this will make them grumpy.) Either way, you'll discover integration conflicts before you merge up, which in turn helps keep main clean.
- **Create a pull request when you're ready to merge up.** This means the implementation is completed, you've pulled in changes from your teammates and resolved any resulting conflicts, and all tests against your branch are passing.
- **Merge up, and deploy at will.** Some teams prefer to automatically ship each change as soon as it's merged to main and all the tests pass there – the continuous *deployment* model. Other teams prefer to make a human decision about which changes to ship, and when. It's up to you and your team.

So there you have it. Branching workflows make continuous delivery a simpler proposition, and Git takes the headaches out of branching. Keep reading for deeper dives into setting up your Git repository to be CD-friendly and how to put all these ideas into practice using your Atlassian tools. See you on the next page!

SHARE THIS ARTICLE

Microservices Architecture

Bitbucket CI/CD tutorials

Continuous Delivery articles

SARAH GOFF-DUPONT

I've been involved in software for over 10 years: testing it, automating it, and now writing about it. When not at work, I can be found reading contemporary fiction, smashing it out & keeping it real at CrossFit, or rolling around on the floor with my kids. Find me on Twitter!@DevToolSuperfan

TUTORIAL

Continuous Delivery Tutorial

In this guide, we'll see how you can use Bitbucket Pipelines to adopt a continuous delivery workflow. Read on!

Try this tutorial →

ARTICLE

Why agile isn't agile without continuous delivery

In order to reap the benefits of agile, you need to be agile through all phases of the software development lifecycle.

Read this article →