5 tips for CI-friendly Git repositories (and Git-friendly CI)

Set yourself up for success – it all starts with your repository.

BY SARAH GOFF-DUPONT

Browse topics

Continuous Delivery Principles

Continuous Delivery Pipeline 101

What is Continuous Integration

- Overview
- How to get to Continuous Integration
- How infrastructure as a service empowers the modern enterprise
- 3 Git tools for Continuous Integra
- **5 Tips for CI-Friendly Git Repos**
- Continuous Integration Tools
- Trunk-based Develop

Software testing for continuous delivery

What Is Continuous Deploym

Microservices and Microservice Architect

Bitbucket CI/CD tutorials

Continuous Delivery articles

As I talked about in another article, Git and continuous delivery comprise one of those delicious "chocolate & peanut butter" combinations we occasionally find in the software world – two great tastes that taste great together. So I want to share some tips for getting your builds in Bamboo to play nicely with your Bitbucket repositories. Most of their interaction happens in the build and test phases of continuous delivery, so you'll notice me talk mostly in terms of "CI," rather than "CD."

1: Store large files outside your repo

One of the things you often hear about Git is that you should avoid putting large files into your repository: binaries, media files, archived artifacts, etc. This is because once you add a file, it will always be there in the repo's history, which means every time the repo is cloned, that huge heavy file will be cloned along with it.

Getting a file out of the repo's history is tricky – it's the equivalent of performing a frontal lobotomy on your code base. And this surgical file extraction alters the whole history of the repo, so you no longer have a clear picture of what changes were made and when. All good reasons to avoid large files as a general rule. And...

Keeping large files out of your Git repos is especially important for CI.

Each time you build, your CI server has to clone your repo into the working build directory. And if your repo is bloated with a bunch of huge artifacts, it slows that process down and increases the time your developers have to wait for build results.

Ok, fine. But what if your build depends on binaries from other projects or large artifacts? That's a very common situation, and probably always will be. So the question is: how can we handle it effectively?

An external storage system like Artifactory (who make an add-on for Bamboo), Nexus, or Archiva can help for artifacts that are generated by your team or the teams around you. The files you need can be pulled into the build directory at the beginning of your build – just like the 3rd-party libraries you pull in via Maven or Gradle.

> **Pro tip:** If the artifacts change frequently, avoid the temptation to sync your big files to the build server every night so you only have to transfer them across the disc at build time. In between your nightly syncs, you'll end up building with stale versions of the artifacts. Plus, developers need these files for builds on their local workstations anyway. So overall, the cleanest thing to do is to just make artifact download part of the build.

If you don't already have an external storage system on your network, it's easiest to take advantage of Git large file support (LFS).

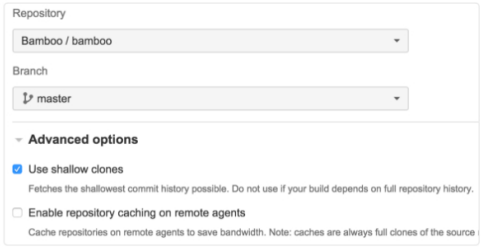
Git LSF is an extension that stores pointers to large files in your repository, instead of storing the files themselves. The files themselves are stored on a remote server. As you can imagine, this drastically reduces clone time.

Chances are, you already have access to Git LFS – both Bitbucket and Github support it.

2: Use shallow clones for CI

Each time a build runs, your build server clones your repo into the current working directory. As I mentioned before, when Git clones a repo, it clones the repo's entire history by default. So over time, this operation will naturally take longer and longer. Unless you enable shallow cloning in Bamboo.

With shallow clones, only the current snapshot of your repo will be pulled down. So it can be quite useful for reducing build times, especially when working with large and/or older repositories.



But let's say your build requires the full repo history – if, for example, one of the steps in your build updates the version number in your POM (or similar), or you're merging two branches with each build. Both these cases require Bamboo to push changes back to your repo.

As of Git 1.9, simple changes to files (like updating a version number) can be pushed without the entire history present. But merging still requires the repo's history because Git needs to look back and find the common ancestor of the two branches – that's going to be a problem if your build uses shallow cloning. Which leads me to tip #3.

3: Cache the repo on build agents

This also makes the cloning operation much faster, and Bamboo actually does this by default.

Note that repo caching only benefits you if you're using agents that persist from build to build. If you create and tear down build agents on EC2 or another cloud provider every time a build runs, repo caching won't matter because you'll be working with an empty build directory and will have to pull down a full copy of the repo every time anyway.

Shallow clones plus repo caching, divided by persistent vs. elastic agents, equals an interesting web of factors. Here's a little matrix to help you strategize.

		Shallow clones	Repo caching
PERSISTENT AGENTS	No changes	✓	✓
	Simple changes	✓	✓
	Auto-merging	✗	✓
ELASTIC AGENTS	No changes	✓	✗
	Simple changes	✓	✗
	Auto-merging	✗	✗

4: Choose your triggers wisely

It goes (almost) without saying that running CI on all your active branches is a good idea. But is it a good idea to run

all builds on all branches against all commits? Probably not. Here's why.

Let's take Atlassian, for example. We have upwards of 800 developers, each pushing changes to the repo several times a day – mostly pushes to their feature branches. That's a lot of builds. And unless you scale your build agents instantly and infinitely, it means a lot of waiting in the queue.

One of our internal Bamboo servers houses 935 different build plans. We plugged 141 build agents into this server, and used best practices like artifact passing and test parallelization to make each build as efficient as possible. And still: building after each and every push was clogging up the works.

Instead of simply setting up another Bamboo instance with another 100+ agents, we stepped back and asked if this was truly necessary. And the answer was no.

So we gave the developers the option of making their branch builds push-button instead of always triggering then automatically. It's a good way to balance testing rigor with resource conservation, and branches are where most of the change activity is happening, so there's a big opportunity for savings.

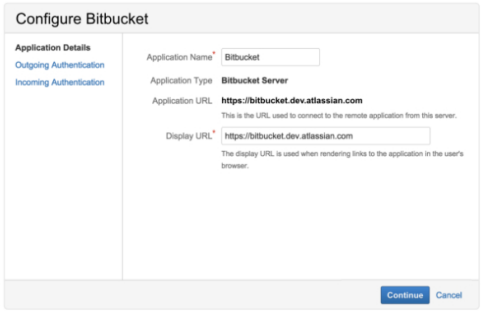
Many developers like the extra control that push-button builds offer, and find fit naturally into their workflow. Others prefer not to think about when to run a build, and stick with automated triggers. Either approach can work. The important thing is to get your branches under test in the first place, and make sure you've got a clean build before merging upstream.

Critical branches like main and stable release branches are a different story, though. Builds there are triggered automatically, either by polling the repo for changes or by sending a push notification from Bitbucket to Bamboo. Since we use dev branches for all our work-in-progress, the only commits coming into main should (in theory) be dev branches getting merged in. Plus, these are the code lines we release from and make our dev branches from. So it's really important that we get timely test results against each merge.

5: Stop polling, start hooking

Polling your repo every few minutes looking for changes is a pretty cheap operation for Bamboo. But when you're scaling up to hundreds of builds against thousands of branches involving dozens of repos, it adds up fast. Instead of taxing Bamboo with all that polling, you can have Bitbucket call out when a change has been pushed and needs to be built.

Typically, this is done by adding a hook to your repository, but as it happens, the integration between Bitbucket and Bamboo does all the under-the-hood set-up for you. Once they're linked on the back end, repo-driven build triggers Just Work™ right out of the box. No hooks or special configs required.



Regardless of tooling, repo-driven triggers carry the

advantage of automatically fading into the sunset when the target branch goes inactive. In other words, you'll never