radar.oreilly.com

Continuous deployment in 5 easy steps

Eric Ries

11-14 minutes

Print

One of the lean startup techniques I'll be discussing at this week's session at the Web 2.0 Expo is called continuous deployment. It's a process whereby all code that is written for an application is immediately deployed into production. The result is a dramatic lowering of cycle time and freeing up of individual initiative. It has enabled companies I've worked with to deploy new code to production as often as fifty times every day.

Continuous deployment is controversial. Most people, when they first hear about continuous deployment, think I'm advocating low-quality code or an undisciplined cowboy-coding development process. On the contrary, I believe that continuous deployment requires tremendous discipline and can greatly enhance software quality, by applying a rigorous set of standards to every change to prevent regressions, outages, or harm to key business metrics. (This criticism is a variation of the "time, quality, money – pick two" fallacy)

Another common reaction I hear to continuous deployment is that it's too complicated, time-consuming, or hard to prioritize. It's this latter fear that I'd like to address head-on in this post. While it is true that the full system we use to support deploying fifty times a day at IMVU is elaborate, it certainly didn't start that way. By making a few simple investments and process changes, any development team can be on their way to continuous deployment. It's the journey, not the destination, that counts. Here's the why and how, in five steps.

1. Continuous integration server. This is the backbone of continuous deployment. We need a centralized place where all automated tests (unit tests, functional tests, integration tests, everything) can be run and monitored upon every commit. There are many fine free software tools to make this easy – I have had success with BuildBot. Whatever tool you use, it's important that it be able to run all the tests your organization writes, in all languages and frameworks.
   If you only have a few tests (or even none at all), don't despair. Simply set up the CI server and agree to one simple rule: we'll add a new automated test every time we fix a bug. Following that rule will start to immediately get testing where it's needed most: in the parts of your code that have the most bugs and, therefore, drive the most waste for your developers. Even better, these tests will start to pay

immediate dividends by propping up that most-unstable code and freeing up a lot of time that used to be devoted to finding and fixing regressions (aka “firefighting”).

If you already have a lot of tests, make sure that the total time the CI server spends on a full run is a small amount of time, 10-30 minutes at the maximum. If that’s not possible, simply partition the tests across multiple machines until you get the time down to something reasonable.

For more on the nuts-and-bolts of setting up continuous integration, see Continuous integration step-by-step.

2. Source control commit check. The next piece of infrastructure we need is a source control server with a commit-check script. I’ve seen this implemented with CVS, subversion or Perforce and have no reason to believe it isn’t possible in any source control system. The most important thing is that you have the opportunity to run custom code at the moment a new commit is submitted but before it is accepted by the server. Your script should have the power to reject a change and report a message back to the person attempting to check in. This is a very handy place to enforce coding standards, especially those of the mechanical variety.
   But its role in continuous deployment is much more important. This is the place you can control what I like to call “the production line” to borrow a metaphor from manufacturing. When something is going wrong with our systems at any place along the line, this script should halt new commits. So if the CI server runs a build and even one

test breaks, the commit script should prohibit new code from being added to the repository. In subsequent steps, we'll add additional rules that also "stop the line," and therefore halt new commits.

This sets up the first important feedback loop that you need for continuous deployment. Our goal as a team is to work as fast as we can reliably produce high-quality code – and no faster. Going any "faster" is actually just creating delayed waste that will slow us down later. This feedback loop is also discussed in detail elsewhere.

3. Simple deployment script. At IMVU, we built a serious deployment script that incrementally deploys software machine-by-machine and monitors the health of the cluster and the business along the way so that it can do a fast-revert if something looks amiss. We call it a cluster immune system. But we didn't start out that way. In fact, attempting to build a complex deployment system like that from scratch is a bad idea.
   Instead, start simple. It's not even important that you have an automated process, although as you practice you will get more automated over time. Rather, it's important that you do every deployment the same way and have a clear and published process for how to do it that you can evolve over time.

   For most websites, I recommend starting with a simple script that just rsync's code to a version-specific directory on each target machine. If you are facile with unix symlinks, you can pretty easily set this up so that advancing to a new version (and, hence, rolling back) is as easy as switching a single symlink on each server. But even if that's not appropriate for your setup, have a single script that does a deployment directly from source control.

When you want to push new code to production, require that everyone use this one mechanism. Keep it manual, but simple, so that everyone knows how to use it. And, most importantly, have it obey the same "production line" halting rules as the commit script. That is, make it impossible to do a deployment for a given revision if the CI server hasn't yet run and had all tests pass for that revision.

4. Real-time alerting. No matter how good your deployment process, bugs can still get through. The most annoying variety are bugs that don't manifest until hours or days after the code that caused them is deployed. To catch those nasty bugs, you need a monitoring platform that can let you know when things have gone awry, and get a human being involved in debugging them.
   To start, I recommend a system like the open source nagios. Out of the box, it can monitor basic system stats like load average and disk utilization. For continuous deployment purposes, we want to be able to have it monitor business metrics like simultaneous users or revenue per unit time. At the beginning, simply pick one or two of these metrics to use. Anything is fine to start, and it's important not to choose too many. The goal should be to wire the nagios alerts up to a pager, cell phone, or high-priority email list that will wake someone up in the middle of the night if one of these metrics goes out of bounds. If the pager goes off too often, it won't

get the attention it deserves, so start simple.

Follow this simple rule: every time the pager goes off, halt the production line (which will prevent checkins and deployments). Fix the urgent problem, and don't resume the production line until you've had a chance to schedule a five whys meeting for root cause analysis, which we'll discuss next.

5. Root cause analysis (five whys). So far, we've talked about making modest investments in tools and infrastructure and adding a couple of simple rules to our development process. Most teams should be able to do everything we've talked about in a week or two, at the most, since most of the work is installing and configuring off-the-shelf software.
Five whys is not something you can get in a box. It's a powerful practice that is the motive force that will drive major improvements in development process incrementally, one step at a time. I've described it in detail in my post Five Whys, and will only summarize it here.

   The idea is to always get to the root cause of any unexpected failure in the system. A test failing, a nagios alert firing, or a customer seeing a new bug are all sufficient triggers for root cause analysis. That's why we always shut down the production line for problems of this kind – it signals the need for root cause analysis and also creates the time and space for it to happen (since it deliberately slows the whole team down).

   Five whys gets its name from the process of asking "why" recursively to uncover the true source of a given problem.

The way five whys works to enable continuous deployment is when you add this rule: every time you do a root cause analysis, make a proportional investment in prevention at each of the five levels you uncover. Proportional means that the solution shouldn't be more expensive than the problem you're analyzing; a minor inconvenience for only a few customers should merit a much smaller investment than a multi-hour outage.

But no matter how small the problem, always make some investments, and always make them at each level. Since our focus in this post is deployment that means always asking the question "why was this problem not caught earlier in our deployment pipeline?" So if a customer experienced a bug, why didn't nagios alert us? Why didn't our deployment process catch it? Why didn't our continuous integration server catch it? For each question, make a small improvement.

Over months and years, these small improvements add up, much like compounding interest. But there is a reason this approach is superior to making a large up-front investment in a complex continuous deployment system modeled on IMVU's (or anyone else's). The payoff is that your system will be uniquely adapted to your particular system and circumstance. If most of your headaches come from performance problems in production, then you'll naturally be forced to invest in prevention at the deployment/alerting stage. If your problems stem from badly factored code, which causes collateral damage for even small features or fixes, you'll naturally find yourself adding a lot of automated tests to your CI server. Each problem drives investments in that category of solution. Thankfully there's an 80/20 rule at work: 20% of your code and architecture probably drives 80% of your headaches. Investing in that 20% frees up incredible time and energy that can be invested in more productive things.

Following these five steps will not give you continuous deployment overnight. In its initial stages, most of your root cause analysis will come back to the same problem: "we haven't invested in preventing that yet." But with patience and hard work, anyone can use these techniques to inexorably drive waste out of their development process.

Continuous deployment is only one tool in the lean startup arsenal – to learn more about it and to see how it relates with other techniques, join us for "The Lean Startup: a Disciplined Approach to Imagining, Designing, and Building New Products" on April 1st at Web 2.0 Expo. And thanks to web2open, you can also join for an in-depth discussion immediately afterward. Best of all, you can attend both events for free (click here for details).

tags: