SUBJECT7
QUALITY IS A CULTURE

Continuous Deployment Explained

By Matthew Heuser for Subject7

If you have a secret hobby of arguing over words, or perhaps worse, injecting meaning that is incorrect into a technical term, well ... software might be the place for you. Consider *Quality Assurance*, or QA, which is something often used by people who do not have change permissions on the version control system, nor ultimate decision on what bugs should block release. Or "capital-v" *Verification* and *Validation*, two words that, in terms of "checking on some piece of work to see if it is good," mean pretty much the same thing. The military defense acquisition industry has made these two entirely different disciplines, claiming that validation is "building the right thing", while verification is "building the *thing* right." No, you read that correctly. In order to get even more absurd, the hair-splitters use the very same words to make the distinction.

Those examples may just pale in comparison to the "delivery" and "deployment." After all, isn't delivery to put things in production while deployment is, well, to put things in production? Today I'll provide a distinction that makes the issue clear, go into *continuous deployment* in a little bit of detail, and, perhaps at the end, provide some justification for the term.

Defining Continuous Delivery and Deployment

The academic definition of *continuous delivery* is that teams produce software in short cycles, that the software can be released at any time, and that release is done manually, typically by a push or click. That sounds like it could be a mini-waterfall, but things are a little more complex than that. With continuous delivery, each micro-feature goes through its own automated build, automated check, and deployment to a test environment. At that point, a human can explore the feature, which now lives on a staging server, along with any other manual steps. Finally, someone can push just that change to production. With continuous delivery, there is no "retest everything" human step, no "regression testing" process done by people. The team then has the engineering tools to release just that feature without risking the entire release.

The classic way to do continuous delivery was to use PHP, where each feature was essentially its own web page. As long as the programmers did not need to change the database or the code libraries, a change to a file could only break that one page. This meant a developer could change any single page (even the "home page"), roll it to test, explore it on test, and roll to production. Etsy, one of the earliest published examples, was the poster child for this, pioneering innovations in monitoring and correction that allowed problems to be found and fixed quickly.

Continuous deployment, as Timothy Fitz defined it, skips that manual step. With continuous deployment, if the build passes, the code rolls to production. The key is that the "roll to production" needs to be responsible and reliable.

Again, Etsy was an example of this approach -- knowing that a change would be localized to a single page. Of course, code that touched credit card or personal customer information would go through a different process with more checks. In addition, Etsy popularized the feature flag, a way to slowly release code to larger and large groups.

In simple terms, a feature flag allows you to role the code out to just a select user group. That means the code looks a bit like this:

```
if (Has_Feature("FeatureName",GetUserID()) {
   //New Code Here

}
```

Feature flags allows a team to roll out a change to just a select group of users. That means the code can roll to production, but only the testers can see the new functionality. The testers can test on production. If the code works, changing the feature flag is typically a button-push process. If not, the programmers can fix the code. That allows the team to "push" code to production without much less risk.

Notice I wrote *less* risk, not *no* risk.

Risks of Continuous Deployment

Feature flags tend to introduce large amounts of extra "if" statements. Over time, these become ifs nested within ifs within ifs. It is possible the programmer forgets one of those if statements, or mistypes, or makes some error in promoting or demoting feature flags. It is even possible that having feature flagA on while feature flagB is off creates an unintended scenario, and a rollback of one but not the other creates a problem in production. All of these "if" statements create a fair bit of new code and new combinations that are likely untested as combinations.

There are software engineering ways to deal with this - pushing those conditional "if" statements into classes and using design patterns. That creates even more code, and it becomes unlikely there will be unit tests to cover the combination. Test automation tools typically don't toggle configuration flags to test them either.

Most of the time, most feature flags are isolated. Some teams go through and "clean up" old toggles that "should" always now be on. Fundamentally, the risk of continuous delivery is in the definition. The entire system, end to end, pushes to production on commit if all the automated, computer-generated checks pass. The Etsy way, to trust some features to this sort of release while having more human quality gates for things that are core or cash, might be the best way for most companies to get started.

But ... why the silly name?

People have known about the silliness of "QA" for quality assurance for a long time. In my experience, no thinking executive really believes "QA" can assure quality. The few that haven't thought it through are corrected soon enough with no harm done. The term is flawed, but not fatally so. When I go into an organization that has a "QA Department," correcting people and fixing the department name is rarely on my list.

That is how I feel about continuous deployment. It takes a little explanation; you need to point people to an article like this one. On the whole, it is an innovation that has many benefits and a few risks. Even if I wanted to get rid of the term, there comes the question of what to call it instead. Years ago, the inventors of *extreme programming* were worried about the risks of the term "Perfect Engineering Days," so they invented Gummi Bears. While actual gummi bears are sticky, the idea did not stick until it became known as Velocity. In Velocity, we have a word whose general meaning matches what we really mean.

We might not find a match like that for continuous deployment. For now, we'll be forced to have some of the same conversations, over and over, to build shared understanding.

The alternatives to that might be inventing something else, perhaps stealing the name of a vegetable, or using a term from rugby.

For now, *continuous deployment* should do just fine.