Staging Servers Must Die

edith harbaugh / 22 Jan 2016 / Code / Connected Devices / Fintech / Health / Industrial / Smart Cities / Uncatogorized

readwrite

Guest author Edith Harbaugh is the CEO and cofounder of LaunchDarkly, a service that helps software teams launch, measure, and control their features.

The DevOps revolution has focused on automation and smoothing the process of getting code from developer box to production. Essential tools like Puppet, Ansible, and Chef enable builds in minutes where it used to take hours.

DevOps 1.0 has made it easier to maintain the traditional waterfall pattern of developer box, QA, staging, and production. However, even the notion of a build is increasingly antiquated. If developers can check code in every day or every hour, the idea of a build is antiquated, as are separate systems for QA, staging, and production. Maintaining these separate systems is a legacy of waterfall development, and will be subsumed by true agile deployment and what I am calling "DevOps 2.0."

Hanging Separately

When software development first began, code changes were immediately checked into the production environment. There were many issues with this. If something broke, it was immediately broken for everyone. If you had multiple developers (or multiple teams of multiple developers) the chance of something breaking climbed exponentially. Though it was time-efficient to move directly into production, the price was paid with quality and stability.

A best practice arose of "cascading" environments and moving releases though a release cycle of separate servers for development, test, staging, and production. These separate servers tried to map into the roles in a good software organization. Product managers and user experience design the product, making requirements and interface. Developers code the requirements. Quality assurance verifies that the code does its job and doesn't break any old functions.

Depending on the size of the organization, each role could have multiple people or teams, or just one person doing all roles. But even then, the idea of separation ruled.

Cascading release servers increased quality and stability of code, but at the expense of time. Developers would code on their own local boxes, checking into a master source-control system. When enough changes were made, a build would be cut and put on a quality-assurance Server. Quality-assurance engineers would verify the release on QA, and document any issues for developers with old code or new. The developers would address the issues, then push a new build to QA.

When QA had verified the build to satisfaction, the build would then be put on staging. Staging was intended to be as close to production as possible, in terms of actual user data. On staging, product managers might give a final check of their features, and performance engineers might run load tests.

There are four key issues with this workflow:

Time

Even super-quick deployments take 5–15 minutes. If an issue is found in production, it can take hours to propagate a fix. The fix must first be verified on QA, and all regression tests done. Repeat on staging, and then push to production.

Inability To Keep Systems In Sync

From my time at Epicentric, I have a patent on packaging up components to push through different systems. Having a separate QA server, staging server, and production server means maintaining three separate systems. Usually an attempt is made to have staging and production have as close as possible the same data set, while QA is a free for all.

In practice, it makes bugs sometimes very difficult to replicate. Many times a bug can't be verified on QA as the data set is different. In addition, it's very easy to get confused.

Expense

Simulating production loads on staging can be extremely expensive, with bills of $3,000–$30,000 not uncommon.

Confusion

This is always the biggest risk. It's a pain to get internal users to use staging as a real system, knowing that their data can always be overridden or forgotten. At TripIt, I'd put trips into staging, knowing that I'd have to recreate my real trips on production. At PlantSense, we ran a separate beta server, and users would always accidentally go to www.easybloom.com rather than beta.easybloom.com, no matter how many times we asked beta users to use a bookmark .This caused confusion and dissatisfaction—beta users would report "missing data" which turned out to be them accessing the www machine, instead of beta where their data was.

A Flag Of Hope

So how do you push directly from development to production?

The answer is feature flags, sometimes known as feature toggles. These are bits of code written into your application that let you turn features on or off at a moment's notice for specific segments of your user base.

A feature flag can encapsulate a given change. You then grant access to the people who need it. The same flow from QA to staging to production can happen—all on the same server. You can use it to test new code with just internal users; a small set of beta users; or a cohort of live users. Etsy's Feature API on GitHub has some interesting examples of how feature flagging can be used.

Feature flagging has the potential to save time and money, increase customer satisfaction, and increase stability and quality. Facebook is well-known for its use of feature flagging, as are many other large tech companies.

Here's what DevOps 2.0 with feature flagging looks like. A developer makes a change, and tests it on their local machine. She then pushes the change to production, wrapped with a feature flag. The feature is off for everyone. QA verifies the feature on production. The product manager verifies that the feature works as expected. Performance can throttle up and down who has access to the feature. There's no need to replicate load—the load on the feature is the real load.

It's even easy to give access to external users like early beta customers, journalists, and industry analysts. Just grant the access to the feature, directly on production.

According to Douglas Squirrel, consulting CTO, the secret to releasing directly to production is "to slice features into tiny, user-ready, deployable pieces—and of course you will want to control access to those pieces with feature flags."

The advantages of pushing directly to production are very clear. Consumers get direct access to features, without the time lag of waiting for them to move through multiple different environments. A huge benefit to an engineering organization is a reduction in the burden and expense of maintaining separate datasets for QA, staging, and production. The tedium wasted in verifying issues in different environments disappears, as well as the time spent keeping them in sync.

"In the promised land, every developer habitually releases finished code to production at least once daily—and when you get there, there are unexpected benefits to code quality and engineering motivation, as well as the faster user feedback you were probably expecting," says Squirrel.

"QA acts as a gateway to ensure we tested our features before releasing them to customers, however gateways introduce delays," says Sam Hatoum, founder of Xolv.io. "With feature flags, we're able to deploy features to our production environment but hide them from our customers until we are ready. And when we are mostly ready, we can use our users that opted-in to our beta program to helps us with our testing. So by removing our QA environment, we actually increased our ability to increase the quality of our features while benefiting from a speed increase."

Better software, faster releases, happier engineers and customers—are you ready for DevOps 2.0?

Photo by Bob Mical

Tags: #application development #Guest Posts #QA #Staging Servers #testing

Edith Harbaugh

Related Posts

Developing a Secure FinTech App – Best Pr...

5 Digital Health Technologies Helping to Stop t...

Top 20 Virtual Blockchain Speakers

Why Real-Time Data Analysis is Crucial for Heal...