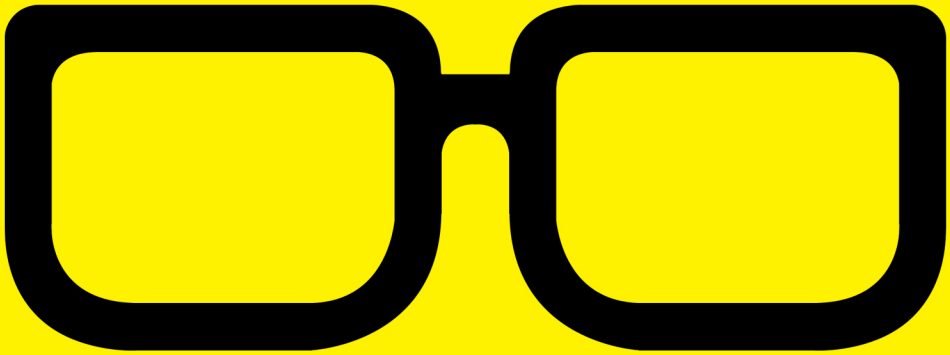
SPONSORED BY

New Relic®

GEEK GUIDE



Slow Down to Speed Up: Continuous Quality Assurance in a DevOps Environment

LINUX™ JOURNAL

Table of Contents

BILL CHILDERS is the Senior Development Operations Manager for MobileIron, a mobile device management company. Bill has worked in IT and DevOps since before the DevOps term was coined, and he has performed a variety of roles in software organizations: systems administrator, technical support engineer, lab manager, IT Manager and Director of Operations. He is the co-author of *Ubuntu Hacks* (O'Reilly and Associates, 2006), and he has been Virtual Editor of *Linux Journal* since 2009. He has spoken at conferences, such as Penguicon and LinuxWorld, and is enthusiastic about DevOps, IT and open source. He blogs at http://wildbill.nulldevice.net and can be found on Twitter at @wildbill.

GEEK GUIDES:
Mission-critical information for the most technical people on the planet.

About the Sponsor

New Relic & DevOps

New Relic is a software analytics company that makes sense of billions of data points about millions of applications in real time. New Relic's comprehensive SaaS-based solution provides one powerful interface for Web and native mobile applications and consolidates the performance monitoring data for any chosen technology in your environment. More than 250,000 active users employ our cloud solution, analyzing more than 200 billion data points across more than 3 million applications every day.

The DevOps movement is focused on helping dev and ops teams improve the odds of application success. New Relic provides the data and accountability application teams need to measure and monitor the impact of new features, prioritize fixes, ensure stability, and keep costs in check. As a DevOps-driven SaaS company, New Relic understands the specific challenges software teams are facing, and has built specific features with agile app delivery in mind.

When your brand and customer experience depend on the performance of modern software, New Relic provides insight into your overall environment. Learn more at **http://newrelic.com**.

Slow Down to Speed Up: Continuous Quality Assurance in a DevOps Environment

BILL CHILDERS
Senior Development Operations Manager for MobileIron
and Virtual Editor of *Linux Journal*

Executive Summary

DevOps is one of the newest and largest movements in Information Technology in the past few years. The name DevOps is a portmanteau of "Development" and "Operations" and is meant to denote a fusion of these two functions in a company. Whether or not your business actually does combine the two functions, the lessons and tools learned from the DevOps movement and attitude can be applied throughout

the entire Information Technology space. This eBook focuses on one of the key attributes of the DevOps movement: Quality Assurance. At any point, you should be able to release your product, code or configuration—so long as you continue keeping your deliverables in a deployable state. This is done by "slowing down" to include a Quality Assurance step at each point in your workflow. The sooner you catch an error or trouble condition and fix it, the faster you can get back on track. This will lower the amount of rework required and keep your team's momentum going in a forward direction, enabling your group to move on to new projects and challenges.

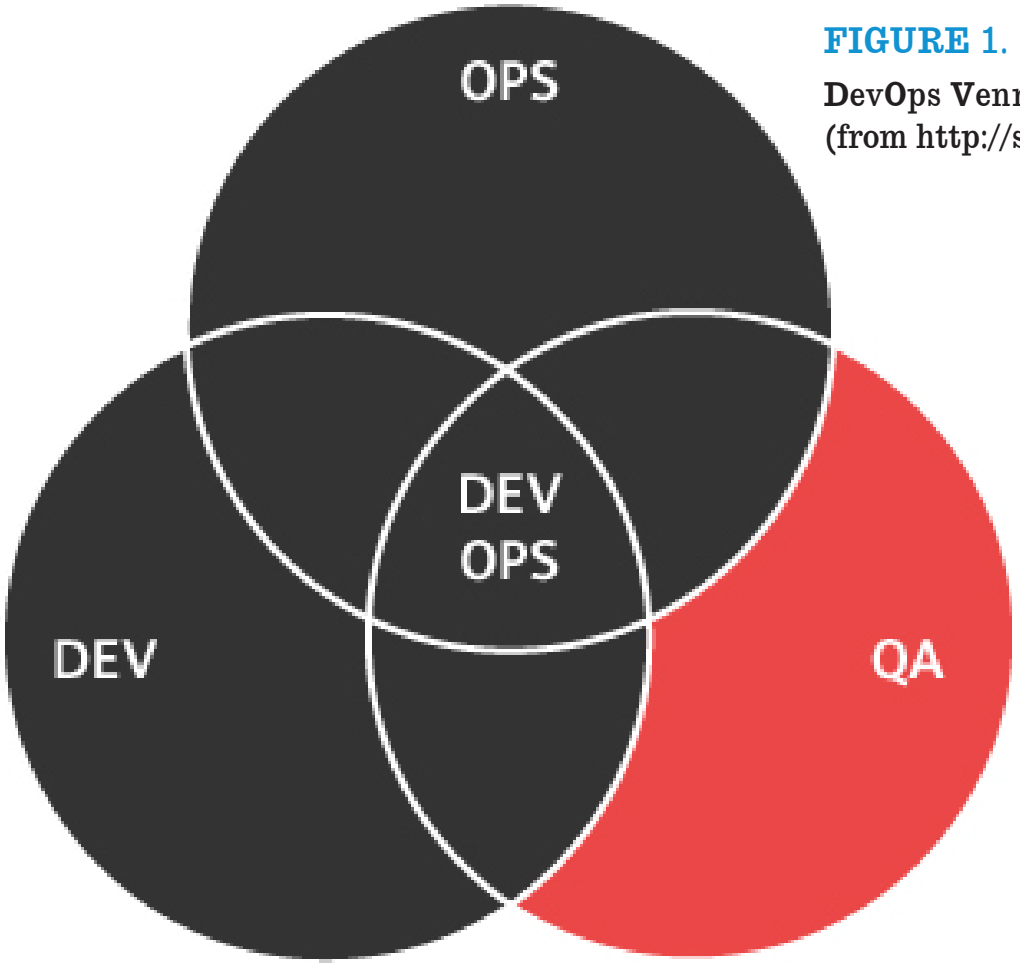


FIGURE 1.
DevOps Venn Diagram (from http://soapui.org)

This eBook dives in to some topics that cross both DevOps and Quality Assurance boundaries. Some of them may apply to you and your organization, and others may not. The goal here is to get you to start thinking about how you can begin injecting Quality Assurance steps into your daily workflow. This eBook also covers some general best practices for a DevOps (or traditional operations organization) and how a Quality Assurance step in conjunction with those best practices can help catch and eliminate errors earlier.

Taking the extra time to do a Continuous Quality Assurance check as a part of your workflow, while it's in flight, may indeed slow you down along the way, but by keeping quality high and rework down to an absolute minimum, you can avoid repeating steps later on—and that's how you speed up over the long haul (see "DevOpsDays Silicon Valley: Continuous Quality" **http://www.serena.com/blog/2013/07/devopsdays-silicon-valley-continuous-quality**).

Process

Let's begin with a talk about the dreaded "P-Word": *process*. For some reason, engineers and other highly technical people cringe when someone mentions the word *process* (particularly if it comes out of a manager's mouth). Perhaps these people had a bad experience with it in an entry-level job or once had a supervisor who was so rigid about following a process that common sense seemed to fly out the window. At any rate, these engineers are among the first to write a program,

script or build automation to ensure reliability and a continuous outcome for some operation or sets of operations, but when asked to apply that same logic to their daily workflow, they rebel. An engineer once said "I don't understand why people fight working through a process—a process is nothing more than a script you execute in meatspace." This definition is precisely what is meant by process: a documented, repeatable series of steps that one follows to ensure the same outcome and keep variables under control.

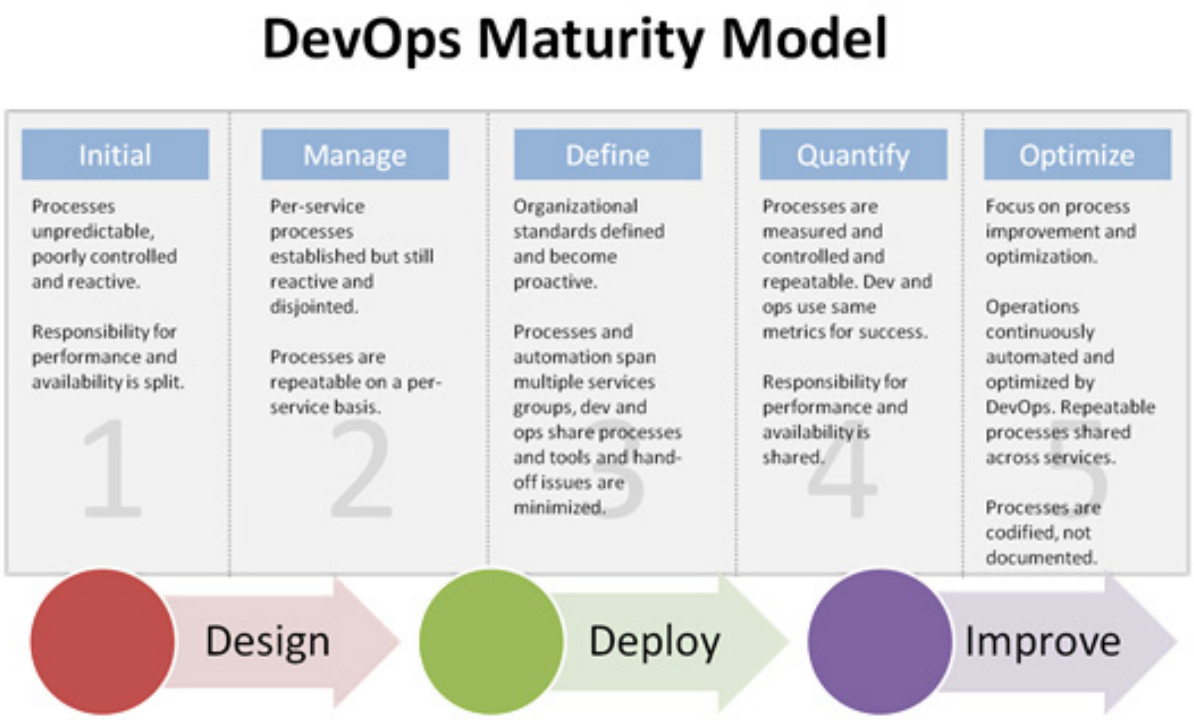


FIGURE 2. DevOps Process Model (from http://devcentral.f5.com)

Your process doesn't need to be a completely rigid, fully automated, push-button, hands-off affair. It may

be though, depending on what's required. On the other hand, it could be something as simple as an internal wiki page that details the steps one needs to follow to perform a particular task. Tailor your process to match the task at hand, as well as the people who will be living within that process. The process will be only as successful as the people executing it, so make sure you've got their input and buy-in. Remember, some people respond very poorly to the word process—so, as a last-ditch effort, don't forget that you can use the word "workflow" as a substitute.

Once you've got your process (or workflow, if you will) sketched out, try to imagine your staff working through the steps. Look for places where errors could be introduced, and then place Quality Assurance steps at those checkpoints. You may be asking yourself, "What kind of Quality Assurance steps could I use?" If you've asked this question, good for you—you're one step ahead. This eBook covers some techniques and tools that can be placed into workflows in the following sections, starting with a frequently overlooked topic and one that's easy to remedy: documentation.

Documentation

Your process or workflow is useless if no one remembers how to execute it. If you were to ask any IT manager if documentation was important, most of them would agree unequivocally. However, there are a *lot* of IT shops running out there with little-to-no documentation, particularly in smaller organizations, where all the knowledge lives in the

brains of one or two people. Any reasonable person quickly would find this to be unacceptable. How do you bring new staff members on board without good documentation? What do you do when an engineer gets sick and another person has to step into that role temporarily?

Here's another case where slowing down can help your organization speed up over the long haul. From the beginning, take the time to document how things work, how to fix things when they go wrong, and what to do at certain points in your process. You can write things down in a wiki, maintain a hard-copy runbook or build in some level of self-documentation to your system. Whatever you do, make it consistent, and continually refer to it and revise it. Here's an opportunity for a Quality Assurance step: keeping it up to date and revised is critical. Documentation that's out of date or incorrect actually may be worse than no documentation, because it can lead you down a false path of believing you're following your process when you're actually *not*. That may be the worst kind of error, as you generally find out you've executed your process incorrectly at the very end, forcing you and your team to re-do everything once more.

Once you have documented your process, you can begin to document other pieces of your environment. A great next step is to document the configurations of all your systems. This could be done initially by placing the configuration files in a Software Configuration Management (SCM) system (more on that later). There are configuration management systems that excel at describing families and groups of systems, which is covered in the next section.

Configuration Management

For a person new to DevOps, it's easy to confuse the large subject of configuration management with DevOps. Most DevOps resources spend a fair amount of time discussing the merits of Puppet versus Chef, or Salt versus Ansible, and it seems like new configuration management systems pop out of the woodwork fairly regularly. Configuration management systems, no matter which you use, simply are tools that describe a system or sets of systems, and they let you reproduce an identically configured system at will. If you've not been exposed to these systems before, each one works by describing a set of attributes that a system or set of systems can have, then they associate those attributes with a system.

All types of system configuration can be described here, from network configuration, to user and SSH key management, to service configuration. This can ensure that all your Web servers run and behave identically or that your mail servers all have SpamAssassin and antivirus services installed and running. Perhaps one of the nicest side benefits of these types of systems is that they check in periodically with a centralized catalog and ensure that the system is continuing to comply and run with the approved configuration. If the system's configuration varies from what's in the catalog, the configuration engine will bring the system back into compliance automatically.

As great as these configuration management systems can be, they're also a place where things can go horribly wrong. What if one of your administrators pushes out a

As great as these configuration management systems can be, they're also a place where things can go horribly wrong. What if one of your administrators pushes out a manifest change to your DNS servers that has a typo in it, causing the DNS service to crash?

manifest change to your DNS servers that has a typo in it, causing the DNS service to crash? That really does happen, and it's a great place for a Quality Assurance step. Many of these systems have a way to test or validate a set of configurations before pushing them to a machine. For instance, on Puppet, the command is `puppet parser validate <filename.pp>`. That could be run by an engineer as a part of your process before continuing on and making that configuration set live. Another possible option is to have a test or "canary" system—something that would run a typical set of services and act as a first line of defense for catching errors in your configuration management system before the error has a chance to make its way out to the larger population of machines.

Tools: From a tools standpoint, there are a lot of different configuration management systems. The current leaders in the space are:

- Puppet from Puppet Labs: **http://puppetlabs.com**

- Chef from Chef Software: **http://www.getchef.com/chef**
- Salt from SaltStack: **http://www.saltstack.com**
- Ansible from Ansible, Inc.: **http://www.ansible.com/home**

Each of these systems has its own unique strengths. If you haven't deployed one of these tools already and are considering doing so, you should do your own research and determine what will work best in your environment. If you're a Ruby shop, Puppet or Chef may be a good fit. If you're proficient in Python, Salt may be more your speed. Like most tools, even the best choice won't automatically ensure you of higher uptimes, better quality or more consistency. Any tool is only as good as the person who wields it.

Proper deployment of a configuration management system can help your operation scale dramatically. Ensuring that what goes into the configuration management system is of the highest quality will help maintain high quality of service, translating into better uptime. Configuration management systems also allow you to scale dramatically—for instance, Facebook uses Chef to manage thousands of servers while keeping its DevOps team relatively small and agile (see "Facebook Uses a Seasoned Chef to Keep Servers Simmering": **http://www.pcworld.com/article/2084900/facebook-uses-a-seasoned-chef-to-keep-servers-simmering.html**).

One way to keep the configurations going into the configuration management system at the optimum

quality is to treat the configurations just like source code. System/service configuration and source code have a lot in common, and many good software development practices can be applied successfully to system configurations too. One of the most basic ways is to check in your configurations to an SCM system or revision control system.

Configuration as Code

Developers have been using SCM systems for ages, and a plethora of them exist. If your business develops any kind of software, you're probably running one of these already, so it may make sense to use whatever SCM your organization has. A possible exception to this *may* be GitHub or another hosted solution, or you may want to control your sensitive and secure data by having the code hosted locally. The bottom line is that you want to treat the system configurations that go into your configuration management system just like source code. In a way, it is source code. It can be "compiled" from a set of files into a running system that performs a desired function, just like source code can be compiled into a running program. This can help keep changes in the environment from being any riskier than they need to be (see "Two DevOps Approaches": http://architects.dzone.com/articles/two-devops-approaches).

SCM Systems: If you're not running any kind of SCM internally, here's a short list of some of the most common ones:

- Subversion (open source): **http://subversion.apache.org**
- Perforce (proprietary): **http://www.perforce.com**
- Git (open source): **http://git-scm.com**
- Mercurial (open source): **http://mercurial.selenic.com**

There is a considerable learning curve to using any of these, but the benefits make the time spent well worth it. Any of these tools can allow multiple people to contribute to a particular project or set of files at once, allowing them to merge their contributions without overwriting one another's work. They also allow you to pinpoint and roll back changes easily—something very useful in the event that an error makes its way into the configuration management tool via the SCM. However, you probably want to try to avoid that error making its way into the SCM at all. How can you do that? This is another great place to slow down and insert a Quality Assurance step, this time in the form of code review.

Code Review

Code review is a fairly simple concept: it boils down to having another person check your work. There are many methods to institute a code review step, from simply passing the proposed configuration change around via e-mail to implementing a code review tool that can enforce a more rigid set of criteria. Whether you decide

A code review tool and step in your process is useless if your personnel bypass it because it's too cumbersome.

to go with a more formal code review or put a more lightweight review step in place, you should make it part of your process that another team member checks all changes before they get committed to the SCM, which then would move them into the configuration management system. Studies show that quick, lightweight code reviews found nearly as many bugs as more formal code reviews did, but were more cost-effective and quicker to perform (see "Best Kept Secrets of Code Review": **http://smartbear.com/SmartBear/media/pdfs/best-kept-secrets-of-peer-code-review.pdf**). The message is the same as for process: tailor the tool to fit your needs, as well as your people. A code review tool and step in your process is useless if your personnel bypass it because it's too cumbersome.

Code Review Tools: Unlike SCM tools, there aren't nearly as many code review tools available, and nearly all of them are open source. Here's a brief list of some of the most common tools:

- Gerrit (open source): **http://code.google.com/p/gerrit**
- Review Board (open source): **http://www.reviewboard.org**

- Crucible: (proprietary, usually bundled with the Fisheye code browser): **http://www.atlassian.com/software/crucible**

All of the tools listed above support a "pre-commit" method of code review, which is desired for this application. What this means is the review occurs *before* the code/configuration is committed to the SCM, and not after, as in a post-commit model. By doing the review with a pre-commit type of workflow, you can ensure that any errors the review Quality Assurance step catches do *not* make it into your server environment.

Although code review can catch many bugs or errors, it won't catch them all. Humans are notoriously undependable in that respect, but between good process with appropriate documentation, a configuration management system where you treat the configuration as code and a good code review process, you should be all set, right? Not exactly, as mistakes still can l eak into your production environment. What's another Quality Assurance step you can take to keep this from happening? You can create a miniature mirror of your production environment—a staging environment, or testbed.

Staging Environments and Monitoring Systems

This may seem like common sense, but it bears saying: if you're making changes directly to your production environment without testing them *somewhere*, you're risking mistakes making their way there as well, harming

Your staging environment doesn't need to be a full mirror of production—it simply needs to have similar characteristics.

your ability to deliver a solid service. The fairly evident answer to this is a mirror of the production environment: a staging environment (see "Continuous Testing and Continuous Monitoring": **http://sdarchitect.wordpress.com/2012/10/30/understanding-devops-part-4-continuous-testing-and-continuous-monitoring**).

Your staging environment doesn't need to be a full mirror of production—it simply needs to have similar characteristics. That's where the beauty of the configuration management system and other pieces of this can help. Just create a new entry in the configuration management system and provision a new machine that looks like production, but with lesser hardware specifications. It even can be as tiny as a single-core virtual machine. It doesn't need a lot of performance; it's simply got to accept and run code.

Once you have a staging environment, you can insert a final Quality Assurance check into your process: require that all changes go into the staging environment first. Ideally, you'd monitor your staging environment with the same monitoring system that you use in production, so you can use all your monitoring checks to ensure that your change didn't break anything. If you don't have

a monitor for that particular change, you should come up with a quick test to check that the change behaves as expected. (This could be an automated unit test, but that's not always practical.) Then, and only then, can that change be promoted to production. Of course, production would be monitored continuously to ensure that all systems are operating optimally.

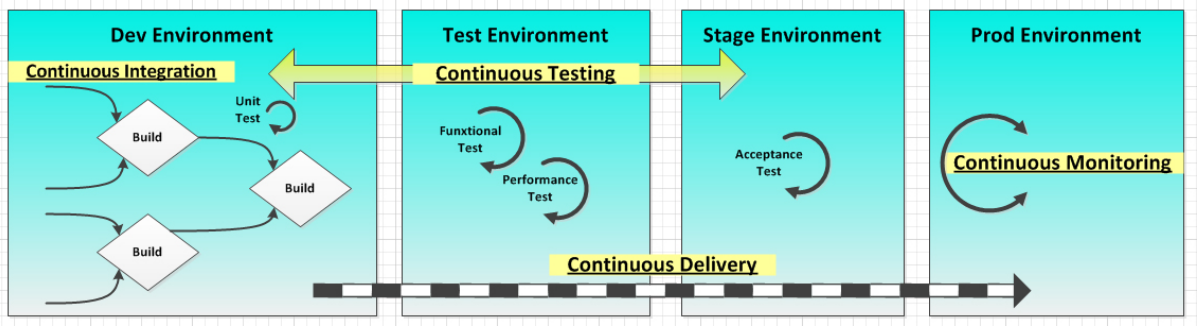


FIGURE 3. Staging Environments and Monitoring (from http://sdarchitect.files.wordpress.com)

Example Workflows

So, what would these workflows look like? Here's an example scenario. Let's say you and your team are responsible for a production e-mail service for awesomecompany.com. This service is configured rather simply, by having a single "mydestination" directive in the /etc/postfix/main.cf file. New top-level domains have been released, and your company wants to receive e-mail at a new domain (awesomecompany.global). Your team has been tasked with configuring the mail servers to accept mail at the new domain in addition to the existing domain (awesomecompany.com). Let's run through the process

twice, first without Quality Assurance steps along the way, and then with the QA steps included—and simulate an error in the mail server configuration in both cases.

The engineer making the change simply needs to make one change to the /etc/postfix/main.cf file.

Original line:

```
mydestination = $myhostname localhost.$mydomain awesomecompany.com
```

The correct, newly edited line should be:

```
mydestination = $myhostname localhost.$mydomain awesomecompany.com
➥awesomecompany.global
```

The engineer actually makes a typo in his change though:

```
mydestination = $myhostname localhost.$mydomain awesomecompany.co
➥awesomecompany.global
```

Note that he accidentally deleted the "m" in .com in the current domain name. Let's see what happens!

Example—No Quality Assurance: The engineer receives the request to add a new domain to the mailserver. The engineer then logs in to the production server, makes the change, saves the /etc/postfix/main.cf file, then reloads the postfix configuration. Now the server is set to receive mail for awesomecompany.global, but due to the typo, the previous domain being accidentally altered, mail that's meant for the main domain gets blocked with a postfix "relay access denied" error, and there's a mail outage in production. All inbound e-mail for awesomecompany.com starts bouncing, and users are no longer getting mail. Since

the mailserver is still up, basic monitoring doesn't report a problem, and the engineer doesn't get word that there's an issue until a user complains. By that time, the engineer's got to spend a fair amount of time troubleshooting the issue until the engineer notices there's a typo.

In summary, awesomecompany.com suffered a mail outage where any inbound e-mail messages were bounced back to their senders—causing several clients to lose confidence in the company and for vital information to be delayed or lost.

Example—With Quality Assurance: The engineer gets the request to add support for the new domain, and the first thing the engineer does is look at the internal wiki to make sure he's going to follow his process for this. Awesomecompany uses Git for its SCM, so the first step in the process is for the engineer to update the local copy of the configuration management system's files. The engineer runs `git pull` to do this. Then, following the process, the engineer makes the change to the /etc/postfix/main.cf file, saving it to his local repository and instituting a code review for the change.

Another engineer in the cubicle next door sees the review request and looks over the configuration line. The second engineer usually is good at spotting typos and ordinarily would catch this, but that engineer's running low on caffeine and the change is not caught.

Now that the code review has been passed erroneously, the engineer who is responsible for the change commits the change to the main Git repository, where the configuration

management system picks up the change and pushes it to the staging environment.

The configuration management system executes the change, and just like in the previous example, mail to awesomecompany.com is bounced. However, the engineer sent an e-mail from his machine to "test@staging.awesomecompany.com", as well as "test@staging.awesomecompany.global", and noticed that the mail to awesomecompany.com was returned as a bounced e-mail.

The engineer then looks over the configuration change and realizes there was a typo. The typo is fixed in short order, and another code review is launched, where the second engineer realizes *his* mistake as well, and that review is passed. The first engineer commits his change to the repository again, where the configuration management system pushes it to the staging environment. The first engineer then sends another two e-mail messages, and both are received in the test mailbox. The engineer then looks at the process document on the wiki again and runs the script to promote the change into production, where it works flawlessly.

Workflow Summary: As you can see, there's quite a few more steps in the workflow that contains the Quality Assurance steps. Each step takes slightly more time to execute, and in the case of the code review step, it requires another engineer's time. However, the fatal production outage is avoided completely, despite the fact that both engineers made at least one mistake.

This is what "slowing down to speed up" means—the

company's overall productivity has not suffered in spite of the fact that two mistakes were made. The accuracy achieved is well worth the extra few minutes those steps required to run.

Conclusion

There isn't a hard and fast way to implement the Quality Assurance concepts outlined in this eBook. Rather, it's left to you and your staff to decide what's appropriate and effective. If you continually "close the loop" and continue to refine and improve your process—including each Quality Assurance step you inject into the process—you'll wind up with a much more stable system overall, and you'll notice your team isn't running around fixing self-inflicted emergencies.■