DevOps Advantages for Testing

Increasing Quality through Continuous Delivery

Gene Gotimer, Coveros
Thomas Stiehm, Coveros

Abstract. DevOps and continuous delivery can improve software quality and reduce risk by offering opportunities for testing and some non-obvious benefits to the software development cycle. By taking advantage of cloud computing and automated deployment, throughput can be improved while increasing the amount of testing and ensuring high quality. This article points out some of these opportunities and offers suggestions for making the most of them.

DevOps

DevOps is a software development culture that stresses collaboration and integration between software developers, operations personnel, and everyone involved in the design, creation, development, and delivery of software. It is based on the same principles that were identified in the Agile Manifesto [1], but while many agile methodologies focus on development only, DevOps extends the culture to the entire software development lifecycle.

Central to DevOps is continuous delivery: delivering software often, possibly multiple times each day, using a delivery pipeline through testing stages that build confidence that the software is a viable candidate for deployment. Continuous delivery (CD) is heavily dependent on automation: automated builds, testing, and deployments. In fact, the reliance on automated deployment is so key that DevOps and CD are often erroneously considered synonymous with automated deployment.

Having a successful delivery pipeline means more than just adding automation. To be effective, tests of all types must be incorporated throughout the process in order to ensure problems aren't slipping through. Those tests include quality checks, functional testing, security tests, performance assessments, and any other type of testing you require before releasing your software.

The delivery pipeline also opens up opportunities to add more testing. Static analysis tools can review code style and test for simple security errors. Automated deployments allow automated functional testing, security tests of the software system as deployed, and performance testing on production-like servers. Continuity of operations (COOP) plans can be tested every time that the infrastructure changes, not just annually in front of the auditors.

With this additional testing, CD can produce software that has fewer defects, can be deployed more reliably, and can be delivered far more confidently than traditional methodologies. Escaped defect rates drop, teams experience lower stress, and delivery is driven by business need. The benefits aren't just slight improvements. In fact, a 2015 report on DevOps from Puppet Labs found that teams using DevOps experience "60 times fewer failures and recover from failures 168 times faster than their lower-performing peers. They also deploy 30 times more frequently with 200 times shorter lead times [2]."

The choices of tools and frameworks for all of this automation has grown dramatically in recent years, with options available for almost any operating system, any programming language, open source or commercial, hosted or as-a-service. Active communities surround many of these tools, making it easy to find help to start using them and to resolve issues.

Continuous Integration

Building a CD process starts with building a Continuous Integration (CI) process. In CI developers frequently integrate other developer's code changes, often multiple times a day. The integrated code is committed to source control then automatically built and unit tested. Developers get into the rhythm of a rapid "edit-compile-test" feedback loop. Integration errors are discovered quickly, usually within minutes or hours of the integration being performed, while the changes are fresh on the developer's minds.

A CI engine, such as Jenkins [3], is often used to schedule and fire off automated builds, tests, and other tasks every time code is committed. The automated build for each commit makes it virtually impossible for compilation errors and source code integration errors to escape unnoticed. Following the build with unit tests means the developers can have confidence the code works the way they intended, and it reduces the chance that changes had unintended side effects.

Important: Continuous integration is crucial in providing a rapid feedback loop to catch integration issues and unintended side effects.

The choice of CI engine is usually driven by the ecosystem you are working in. Common choices include Jenkins for Linux environments and Team Foundation Server [4] for Windows environments.

Code Coverage

CI can also tie-in code coverage tools that measure the amount of code that is executed when the unit tests are run.

Code coverage can be a good guide as to how well the code is unit tested, which in turn tells you how easy it should be to reorganize the code and to change the inner workings without changing the external behavior, a process known as refactoring [5]. Refactoring is an important part of many agile development methodologies, such as extreme programming (XP) [6] and test-driven development (TDD) [7].

In TDD, a test is written to define the desired behavior of a unit of code, which could be a method or a class. The test will naturally fail, since the code that implements the behavior is not yet written. Next, the code is implemented until the test passes. Then, the code is refactored by changing it in small, deliberate steps, rerunning the tests after each change to make sure that the external behavior is unchanged. Another test is written to further define the behavior, and the "test-implement-refactor" cycle repeats.

By definition, code behavior does not change during refactoring. If inputs or outputs must change, that is not refactoring. In those cases, the tests will necessarily change as well. They must be maintained along with, and in the same way as, other source code.

Without sufficient code coverage you cannot be sure that behavior is unchanged. A change in the untested code may have an unintended effect elsewhere. Having enough unit testing and code coverage means you are free to do fearless refactoring: you can change the design and implementation of the software without worrying something will break inadvertently. As the software evolves and you learn more about how the software should have been written you can go back and make changes rather than living with early decisions. In turn, you can move faster at the beginning by "doing the simplest thing that could possibly work [8]" rather than agonizing over every decision to (impossibly) make sure it will address all future needs, known and unknown.

Important: Unit testing and code coverage is about more than just testing. It also enables fearless refactoring and the ability to revisit design and implementation decisions as you learn more.

Code coverage tools are usually programming language-dependent. JaCoCo [9] is an excellent open-source choice for Java, Coverage.py [10] for Python, and NCover [11] is a popular commercial tool for .NET. Every popular programming language today is likely to have several code coverage tool options.

Mutation Testing

Code coverage can't tell the whole story. It only counts how many lines (or methods, or classes, etc.) are executed when the unit tests run, not whether that code is tested well, or at all.

Mutation testing [12] is a process by which existing code is modified in specific ways (e.g., reversing a conditional test from equals to not equals, or flipping a true value to false) and then the unit tests are run again. If the changed code, or mutation, does not cause a test to fail, then it survives. That means the unit tests did not properly test the condition. Even though code coverage may have indicated a method was completely covered, it might not have been completely tested.

Mutation testing generally runs many times slower than unit tests. But if it can be done automatically then the cost of running the mutation tests is only time it takes to review the results. Successful mutation testing leads to higher confidence in unit tests, which leads to even more fearless refactoring.

Suggestion: Use mutation testing tools to determine how effective your unit tests are at detecting code problems.

Tools for mutation testing are available for various programming languages and unit test frameworks. Two mature tools are PIT Mutation Testing [13] for Java and Ninja Turtles [14] for .NET. Humbug [15] is a popular choice for PHP and many options exist for Python [16].

Static Analysis

Static analysis tools are easy to use via the CI engine. These tools handle many of the common tasks of code review, looking at coding style issues such as variable and method-naming conventions. They can also identify duplicate code blocks, possible coding issues (e.g., declared but unused variables), and confusing coding practices (e.g., too many nested if-then-else statements). Having these mundane items reviewed automatically can make manual code reviews much more useful since they can focus on design issues and implementation choices. Since the automated reviews are objective, the coding style can be agreed upon and simply enforced by software.

Important: Static analysis can allow manual code reviews to concentrate on important design and implementation issues, rather than enforcing stylistic coding standards.

Static analysis tools can also identify some serious problems. Race conditions, where parallel code execution can lead to deadlocks or unintended behavior, can be difficult to identify via testing or manual code review, but they can often be detected via static analysis. SQL and other injection vulnerabilities can also be identified, as can resource leaks (e.g., file handle opened but not closed) and memory corruption (e.g., use after free, dangling pointers).

Since static analysis tools can be fast and can easily run automatically as part of the edit-compile-test cycle, they can be used as a first line of defense against coding errors that can lead to serious security and quality issues.

Important: Static analysis tools can provide early detection of some serious code issues as part of the rapid CI feedback cycle.

Every popular programming language has a selection of static analysis tools -- many of them open source. But even easier than choosing one or more and integrating them with your build process or CI engine is installing the excellent open-source tool known as SonarQube [17]. It integrates various analyses for multiple programming languages and displays the combined results in an easy-to-use quality dashboard that tracks trends, identifies problem areas, and can even fail the build when results are beyond project-defined thresholds.

Delivery Pipeline

The delivery pipeline describes the process of taking a code change from a developer and getting it delivered to the customer or deployed into production. CD generally evolves by extending the CI process and adding automated deployment and testing. The delivery pipeline is optimized to remove as many manual delays and steps as practical. The decision to deploy or deliver software becomes a business decision rather than being driven by technical constraints.

The delivery pipeline is often described as a series of triggers: actions such as code being checked into the source control system, that initiate one or more rounds of tests, known as quality gates. If the quality gate is passed, that triggers more processes, which lead to more quality gates. If a quality gate is not passed, the build is not a viable candidate for production, and no further testing is done. The problems that were discovered are fixed and the delivery pipeline begins again.

The delivery pipeline should be arranged so the earliest tests are the quickest and easiest to run and give the fastest feedback. Subsequent quality gates lead to higher confidence that the code is a viable candidate and they indicate more

expensive tests (in regards to time, effort, or cost) are justified. Manual tests migrate towards the end of the pipeline, leaving computers to do as much work as possible before humans have to get involved. Computers are significantly cheaper than people and humans often work slower than computers. They get sidetracked, go to meetings, and don't work around the clock.

The CI process is often the first stage of the delivery pipeline, being the fastest feedback cycle. Often the CI process is blocking: a developer will wait until the quality gate is passed before continuing. Quality gates later in the pipeline are non-blocking: work continues while the quality checks are underway.

While it can be tempting to arrange the delivery pipeline in phases (e.g., unit testing, then functional tests, then acceptance tests, then load and performance tests, then security tests), this leaves the process susceptible to allowing serious problems to progress far down the pipeline, leading to wasted time testing a non-viable candidate for release and extending the time between making a change and identifying any problems. Instead, quality gates should be arranged so each one does enough testing to give confidence the next set of tests is worth doing.

For example, after some functional tests, a quick performance test might be valuable to make sure a change hasn't rendered the software significantly slower. Next, a short security check could be done to make sure some easily detectable security issue hasn't been introduced. Then a full set of regression tests could be run. Later, you could run more security tests along with load and performance testing. Each quality gate has just enough testing to give us confidence the next set of tests is worth doing.

Suggestion: Do just enough of each type of testing early in the pipeline to determine if further testing is justified.

Negative Testing

The first tests written are almost always sunny-day scenarios: does the software do what it was intended to do? We should also make sure there are functions that the software doesn't do: rainy-day scenarios. For example, one user shouldn't be able to look at another user's private data. Bad input data should result in an error message. A consumer should not be able to buy an item if they do not pay. A web-user should not be able to access protected content without logging in. Whenever you identify sunny-day tests, you should also identify related rainy-day tests.

Identifying these conditions while features are being developed will lead to more tests, which will help build more confidence that new features aren't inadvertently introducing security holes. The tests will form a body of regression tests that document how the software is intended to work and not to work. As the code gets more complex, you will be able to fearlessly refactor knowing that you are not introducing unintended side effects.

Important: Sunny-day testing is important, but rainy-day testing can be just as important for regression and security. You need to test both to be confident the code is working correctly.

Automated deployment

Some types of testing aren't valuable until the code is compiled, deployed, and run in a production-like environment. Security scans might depend on the web server configuration. Load and performance tests might need production-sized systems. If deployment is time consuming, error prone, or even just frustrating, it won't be done frequently. That means you won't have as many opportunities to test deployed code.

While an easy, quick, reliable manual install makes it easier to deploy more often, having an automated process can make deployments almost free, especially when deployments can be triggered automatically by passing quality gates. That lets the delivery pipeline progress without human interaction. When there are fewer barriers to deploying, the team will realize there are more chances to exercise the deployment process. When combined with the flexibility of cloud computing resources, deployments will become a regular course of action rather than a step taken only late in the development cycle.

Important: Automated deployments will be used more often than simple manual deployments. They will be tested more often and the delivery pipeline will find more uses for them.

Configuration management tools that perform automated deployments are a class of tool that has garnered a lot of attention in recent years, and many excellent tools, frameworks, and platforms are readily available, both commercially and open source. Puppet [18], Chef [19], and Ansible [20] lead the pack with open-source products that can be coupled with commercial enterprise management systems. Active ecosystems have evolved around each of them with plenty of community support.

Using automated deployments more often gives you more chances to validate that your deployment process works. You can't afford to hope that it works because it runs; you have to verify that it successfully deployed and configured your system or systems using an automated verification process. It has to be quick, so you can afford to run it on each deployment. It should test the deployment, not the application functionality, so focus on the interfaces between systems (e.g., IP addresses and firewalls), configuration properties (e.g., database connection settings), and basic signs of life (e.g., is the application responding). Repeatedly deploying to different environments and then verifying the deployment works gives you higher confidence it will work when deploying to production, which is the deployment that really counts.

Suggestion: Each deployment should be followed with an automated deployment verification suite. Make the deployment verification reusable, so the same checks and tests can be used after each deployment, no matter which environment.

Deployment verification checks can usually be automated using the same tool you use for functional and regression testing. If that tool is too heavyweight or can't be easily integrated into the pipeline, consider a lightweight functional testing framework like Selenium [21] and/or one of the xUnit test frameworks [22], such as JUnit [23] for Java or nUnit [24] for .NET.

Exploratory Testing

Manual exploratory testing is not made obsolete by adopting automated testing. Manual testing becomes more important since automated tests will cover the easy things, leaving the more obscure problems undiscovered. Testers will need increasing amounts of creativity and insight to detect these issues, traits almost impossible to build into automation. The very term exploratory testing highlights the undefined nature of the testing. Automated tests will never adapt to find issues they aren't testing for. This is known as the paradox of automation. "The more efficient the automated system, the more crucial the human contribution [25]."

The delivery pipeline does not have to be an unstopping conveyor belt of releases. Human testers cannot cope with a constant stream of new releases. They cannot deal with the software changing mid-test or even mid-test cycle. Even when they find one problem, there is value in continuing their tests to see if the same problem exists in related functions, or looking for unrelated issues in other parts of the code. There is a balance to make sure time isn't invested testing a non-viable candidate from production and restarting a test suite to fix every little problem individually.

Waiting for human testers to be ready to start a new test cycle slows down the rest of the pipeline. In order to incorporate their testing and not constantly interrupt their test cycles as new versions of the software are made available, consider on-demand deployments, where the pipeline does not deploy to the exploratory testing environment until the testers choose it to be deployed. Or perhaps the software is deployed automatically to a new dynamic environment each time it is packaged, and the testers move on to the most recent (or most important, or most promising) environment. In this way, there is always an environment available for the testers to use without pulling the rug out from under them during their test cycle, thereby buffering the bottleneck [26].

While you want to reduce the time testers spend testing a build that is not viable, you also don't want to start so late as to be a constraint for other activities. Consider running the exploratory testing in parallel with other automated and non-automated tasks, minimizing the wait by placing it at the end of the cycle rather than the start. Think about time boxing (defining and enforcing a fixed duration) the test cycle.

Suggestion: Deployments for manual testing must be coordinated so testers can have a stable environment. Consider on-demand deployments, and make sure the pipeline is only waiting at the end of manual testing, not the beginning.

Parallel Testing

Just as with the exploratory testing, other long-running tests should be run in parallel to make progress while waiting for longer tests to complete. Taking advantage of automated deployments, multiple environments can be built so some tests can be done at the same time using different resources. This can mean doing multiple types of tests at one time, or breaking one type of tests into smaller chunks that can be handled in parallel.

Often four one-day-long tasks are preferable to one four-day-long task because the shorter tasks give additional opportunities for feedback. The fourth day might not need to be needed if there is a show-stopper identified on day three. In parallel, those tests might be run in two parallel tracks, taking a total of two days only. Or perhaps a two-day stress test can be undertaken in parallel with a two-to-three day security scan, to reduce the effect of the bottleneck.

Suggestion: Long-running testing should be done in parallel as much as practical, so that you don't have to wait days or weeks for individual test phases to be completed in sequence.

Infrastructure

Development teams need infrastructure to get their work done. Source code repositories, CI engines, test servers, certificate authorities, firewalls, and issue tracking systems are all examples of tools that might be required, but they are often not deliverables for the project.

Infrastructure doesn't stay static. Systems need to be moved or replicated. They get resized. Applications, tools, and operating systems get upgraded. Hardware goes bad. And other projects need to use the same or similar infrastructure. Setting up your infrastructure is never a one-time occurrence. Even though this infrastructure is internal-facing, it quickly becomes mission critical to the development team.

Treat it like you do production code. Automate the deployment so that redeploying is as easy as pushing out a new version of the software you are writing. Use the same automated deployment tools since you already have experience and tools to support them.

Suggestion: Use your familiarity with the automated deployment tools to automate your infrastructure deployments as well. Treat automated deployment code and infrastructure as mission critical.

Case Study – Forge.mil

DISA's Forge.mil supports collaborative development for the DoD. It is built using commercial off-the-shelf software coupled with open-source tools and custom integration code, written in a variety of programming languages (e.g., Java, PHP, Python, Perl, Puppet). The team used agile techniques from the beginning in order to maximize throughput for the small team doing the integration and development work. The project also served as an exemplar project to demonstrate and document how agile techniques could be used within DoD projects.

An early focus on continuous integration led the team to identify several bottlenecks in the delivery process. Functional testing was manual, slow, and hard to do comprehensively. Development, test, and integration environments were all configured differently from each other and different than production. Deployments were manual, long, complicated, and unreliable. Security patches were often applied directly into production with limited testing, almost always in response to information assurance vulnerability alerts (IAVAs). A team of about two dozen developers, testers, integrators, managers, and others were delivering software to production once every six months. A software release was a big, scary event, carefully planned and scheduled

weeks in advance by the entire team. Problems were identified in the days after each release (often by end users), carefully triaged, with hot fixes deployed or workarounds documented.

The team focused on removing some of these bottlenecks, concentrating on improved functional and regression testing. After discovering the book Continuous Delivery by Jez Humble and Dave Farley [27], they began using Puppet scripts for configuration management which greatly improved the reliability of production deployments. Consistent, production-like deployments in other environments could be performed on-demand in minutes, many times a week. Proactive security testing and vulnerability patching became convenient and did not disrupt other development and testing activities. The bottlenecks the team had identified earlier were eliminated or greatly reduced, one-by-one.

Over time, the team size decreased to less than a dozen people. Software was confidently deployed to production every two weeks with neither drama nor concern. Full regression tests, performance tests, and security tests were regular occurrences multiple times a week. Security patches were incorporated into the normal release cycle, often being fully tested and deployed to production before the IAVAs were even issued. Reports of issues after releases (aka escaped defects) disappeared almost completely. Software releases were driven by business needs and the project management office, not by technical limitations and risks identified by the developers, testers, and integrators.

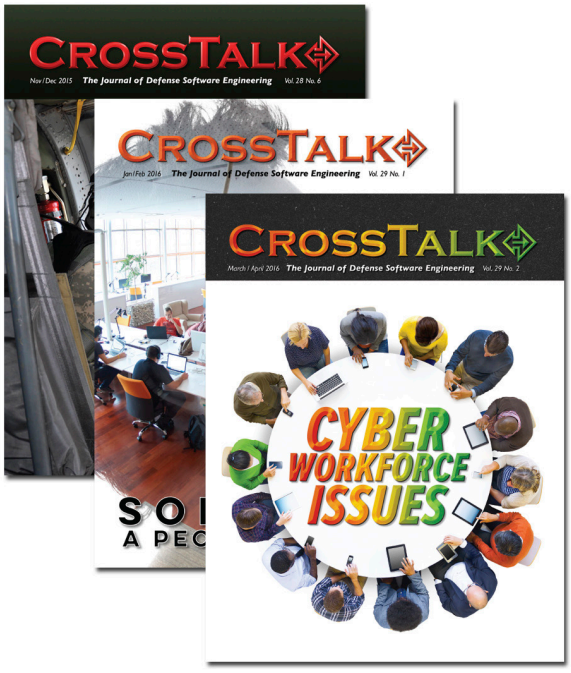
More details are available in Continuous Delivery in a Legacy Shop - One Step at a Time [28], originally presented at DevOps Conference East 2015 in Orlando, Florida.

Conclusion

The journey towards a continuous delivery practice relies heavily on quality tests to show if the software is (or is not) a viable candidate for production. But along with the increased reliance on testing, there are many opportunities for performing additional tests and additional types of tests to help build confidence in the software. By taking advantage of the automated tests and automated deployments, the quality of the software can be evaluated and verified more often and more completely. By arranging the least expensive tests (in terms of time, resources, and/or effort) first, a rapid feedback loop creates openings to fix issues sooner and focus more expensive testing efforts on software that you have more confidence in. By having a better understanding of the software quality, the business can make more informed decisions about releasing the software, which is ultimately one of the primary goals of DevOps.

FURTHER READING

1. Humble, Jez, and David Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Upper Saddle River, NJ: Addison-Wesley, 2011. Print.
2. Kim, Gene, Kevin Behr, and George Spafford. The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win. Portland, Oregon: IT Revolution, 2013. Print.
3. Duvall, Paul M., Steve Matyas, and Andrew Glover. Continuous Integration: Improving Software Quality and Reducing Risk. Upper Saddle River, NJ: Addison-Wesley, 2007. Print.
4. Fowler, Martin, and Kent Beck. Refactoring: Improving the Design of Existing Code. Reading, MA: Addison-Wesley, 1999. Print.



ABOUT THE AUTHORS

Gene Gotimer is a proven senior software architect with many years of experience in web-based enterprise application design, most recently using Java. He is skilled in agile software development as well as legacy development methodologies, with extensive experience establishing and using development ecosystems including: continuous integration, continuous delivery, DevOps, secure software development, source code control, build management, release management, issue tracking, project planning & tracking, and a variety of software assurance tools and supporting processes.

Tom Stiehm has been developing applications and managing software development teams for twenty years. As CTO of Coveros, he is responsible for the oversight of all technical projects and integrating new technologies and application security practices into software development projects. Most recently, Thomas has been focusing on how to incorporate DevOps best practices into distributed agile development projects using cloud-based solutions and how to achieve a balance between team productivity and cost while mitigating project risks. Previously, as a managing architect at Digital Focus, Thomas was involved in agile development and found that agile is the only development methodology that makes the business reality of constant change central to the development process.

REFERENCES

1. Beck, Kent, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. "Principles behind the Agile Manifesto." Agile Manifesto. N.p., 2001. Web. 10 Dec. 2015. <http://agilemanifesto.org/principles.html/>.
2. Puppet Labs 2015 State of DevOps Report. Rep. PwC US, 22 July 2015. Web. 10 Dec. 2015. <https://puppetlabs.com/2015-devops-report>
3. "Welcome to Jenkins CI!" Jenkins CI. CloudBees, n.d. Web. 17 Dec. 2015. <https://jenkins-ci.org/>.
4. "Team Foundation Server." Team Foundation Server. Microsoft, n.d. Web. 19 Jan. 2016. <https://www.visualstudio.com/en-us/products/tfs-overview-vs.aspx>.
5. Fowler, Martin, and Kent Beck. Refactoring: Improving the Design of Existing Code. 1st ed. Reading, MA: Addison-Wesley, 1999. Print.
6. "Extreme Programming." Wikipedia. Wikimedia Foundation, n.d. Web. 21 Jan. 2016. <https://en.wikipedia.org/wiki/Extreme_programming>.
7. "Test-driven Development." Wikipedia. Wikimedia Foundation, n.d. Web. 21 Jan. 2016. <https://en.wikipedia.org/wiki/Test-driven_development>.
8. "Do The Simplest Thing That Could Possibly Work." Do The Simplest Thing That Could Possibly Work. Cunningham & Cunningham, Inc., n.d. Web. 12 Dec. 2015. <http://c2.com/cgi/wiki?DoTheSimplestThingThatCouldPossiblyWork>
9. "JaCoCo Java Code Coverage Library." EclEmma. N.p., n.d. Web. 19 Jan. 2016. <http://eclemma.org/jacoco/>.
10. "Coverage." Python Package Index. N.p., n.d. Web. 19 Jan. 2016. <https://pypi.python.org/pypi/coverage>.
11. "NCover | .NET Code Coverage for .NET Developers." NCover. Gnoso, n.d. Web. 19 Jan. 2016. <http://www.ncover.com/>.
12. "Mutation testing." Wikipedia. Wikimedia Foundation, n.d. Web. 12 Dec. 2015. <https://en.wikipedia.org/wiki/Mutation_testing>
13. "Real World Mutation Testing." PIT Mutation Testing. N.p., n.d. Web. 19 Jan. 2016. <http://pitest.org/>.
14. "NinjaTurtles - Mutation Testing for .NET (C#, VB.NET)." NinjaTurtles. N.p., n.d. Web. 10 June 2015. <http://www.mutation-testing.net/>.
15. "Humbug." GitHub. N.p., n.d. Web. 19 Jan. 2016. <https://github.com/padraic/humbug>.
16. "Index of Packages Matching 'mutationtesting'." Python Package Index. N.p., n.d. Web. 19 Jan. 2016. <https://pypi.python.org/pypi?%3Aaction=search&term=mutation%2Btesting&submit=search>.
17. "Put Your Technical Debt under Control." SonarQube™. SonarSource, n.d. Web. 20 Jan. 2016. <http://www.sonarqube.org/>.
18. "Open Source Puppet." Puppet Labs. Puppet Labs, n.d. Web. 20 Jan. 2016. <https://puppetlabs.com/puppet/puppet-open-source>.
19. "Chef." Chef. Chef Software, Inc., n.d. Web. 20 Jan. 2016. <https://www.chef.io/>.
20. "Ansible Is Simple IT Automation." Ansible. Ansible, Inc., n.d. Web. 20 Jan. 2016. <http://www.ansible.com/>.
21. "Selenium - Web Browser Automation." SeleniumHQ. N.p., n.d. Web. 20 Jan. 2016. <http://www.seleniumhq.org/>.
22. "XUnit." Wikipedia. Wikimedia Foundation, n.d. Web. 20 Jan. 2016. <https://en.wikipedia.org/wiki/XUnit>.
23. "JUnit." JUnit. N.p., n.d. Web. 20 Jan. 2016. <http://junit.org/>.
24. "NUnit." NUnit. N.p., n.d. Web. 20 Jan. 2016. <http://www.nunit.org/>.
25. "Automation." Wikipedia. Wikimedia Foundation, n.d. Web. 17 Dec. 2015. <https://en.wikipedia.org/wiki/Automation#Paradox_of_Automation>
26. Goldratt, Eliyahu M. "Theory of Constraints." Wikipedia. Wikimedia Foundation, n.d. Web. 17 Dec. 2015. <https://en.wikipedia.org/wiki/Theory_of_constraints>
27. Humble, Jez, and David Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Upper Saddle River, NJ: Addison-Wesley, 2011. Print.
28. Gotimer, Gene. "Continuous Delivery in a Legacy Shop - One Step at a Time." SlideShare. Coveros, Inc., 12 Nov. 2015. Web. 20 Jan. 2016. <http://www.slideshare.net/ggotimer/continuous-delivery-in-a-legacy-shop-one-step-at-a-time>.