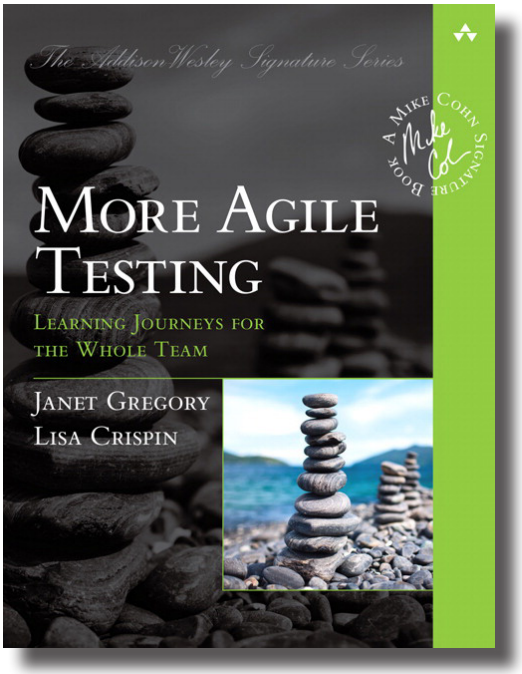
More Agile Testing
Learning Journeys for the Whole Team



544 pages | Paperback | ©2015

Save on Agile Testing

Save 35%* on
- Agile Testing
- More Agile Testing
- Agile Testing eBook Collection

Save 50%* on
Agile Testing Essentials
LiveLessons Video Course

Use code AGILETESTING
at informit.com/agile

More lessons and insights from Janet Gregory and Lisa Crispin, authors of *Agile Testing: A Practical Guide for Testers and Agile Teams*

Packed with new examples from real teams, **More Agile Testing** offers detailed information about adapting agile testing for your environment; learning from experience and continually improving your test processes; scaling agile testing across teams; and overcoming the pitfalls of automated testing. You'll find brand-new coverage of agile testing for the enterprise, distributed teams, mobile/embedded systems, regulated environments, data warehouse/BI systems, and DevOps practices.

- Clarify testing activities within the team
- Collaborate with business experts to identify valuable features and deliver the right capabilities
- Design automated tests for superior reliability and easier maintenance
- Improve and expand Agile team testing skills
- Plan "just enough," balancing small increments with larger feature sets and the entire system
- Use testing to identify and mitigate risks associated with your current agile processes and to prevent defects
- Address challenges within your product or organizational context
- Perform exploratory testing using "personas" and "tours"
- Understand exploratory testing approaches that engage the whole team, using test charters with session- and thread-based techniques
- Bring new agile testers up to speed quickly–without overwhelming them

**Offer only good at informt.com with use of code. during checkout.*
Offer cannot be combined with other offers and is subject to change.

Pearson

informIT
the trusted technology learning source

Praise for *More Agile Testing*

"I love this book. It will help to create really great testers. That's a good thing, since anyone who reads this will want to have one on their team."

—Liz Keogh, agile coach, Lunivore Limited

"This book will change your thinking and move your focus from *tests* to *testing*. Yes, it is not about the result, but about the activity!"

—Kenji Hiranabe, cofounder of Astah and CEO, Change Vision, Inc.

"To my mind, agile development is about learning—that one word captures the true spirit of what agile is all about. When I had the chance to read through their new book, I could only say, 'Wow! Janet and Lisa have done themselves proud.' This is not a book about testing; this is a book about learning. Their clear explanations are accompanied by great true stories and an impressive list of books, articles, and other resources. Those of us who like learning, who love to dig for more information, can rejoice! I know you're always looking for something interesting and useful; I can guarantee that you will find it here!"

—Linda Rising, coauthor of Fearless Change: Patterns for Introducing New Ideas

"Janet and Lisa's first book, *Agile Testing*, drew some general principles that are still important today but left me wondering, 'how?' In this second book, they adapt those principles to today's development landscape—with mobile, DevOps, and cloud-based applications delivered in increasingly compressed release cycles. Readers get specific testing tools for the mind along with new practices and commentary to accelerate learning. Read it today."

—Matt Heusser, Managing Principal, Excelon Development

"An excellent guide for your team's agile journey, full of resources to help you with every kind of testing challenge you might meet along the way. Janet and Lisa share a wealth of experience with personal stories about how they helped agile teams figure out how to get value from testing. I really like how the book is filled with techniques explained by leading industry practitioners who've pioneered them in their own organizations."

—Rachel Davies, agile coach, unruly and coauthor of Agile Coaching

"Let me net this out for you: agile quality and testing is hard to get right. It's nuanced, context-based, and not for the faint of heart. In order to effectively balance it, you need hard-earned, pragmatic, real-world advice. This book has it—not only from Janet and Lisa, but also from forty additional expert agile practitioners. Get it and learn how to effectively drive quality into your agile products and across your entire organization."

—Bob Galen, Principal Consultant, R Galen Consulting Group, and Author of Agile Reflections *and* Scrum Product Ownership

"Janet and Lisa have done it again. They've combined pragmatic life experience with ample storytelling to help people take their agile testing to the next level."

—*Jonathan Rasmusson, author of* Agile Samurai: How Masters Deliver Great Software

"In this sequel to their excellent first book, Janet and Lisa have embraced the maturity of agile adoption and the variety of domains in which agile approaches are now being applied. In *More Agile Testing* they have distilled the experiences of experts working in different agile organizations and combined them with their own insights into a set of invaluable lessons for agile practitioners. Structured around a range of essential areas for software professionals to consider, the book examines what we have learned about applying agile, as its popularity has grown, and about software testing in the process. There is something for everyone here, not only software testers, but individuals in any business role or domain with an interest in delivering quality in an agile context."

—*Adam Knight, Director of QA, RainStor*

"This book has it all: practical advice and stories from the trenches. Whether you've never heard of agile or you think you're an expert, there is something here that will help you out. Jump around in the book and try a few things; I promise you will be a better tester and developer for it."

—*Samantha Laing, agile coach and trainer, Growing Agile*

"*More Agile Testing* is a great collection of stories and ideas that can help you improve not just how you test, but the products you build and the way you work. What I love most about the book is how easy many of the ideas are to try. If one message is clear, it is that regardless of your context and challenges, there are things you can try to improve. Get started today with something small, and nothing will be impossible."

—*Karen Greaves, agile coach and trainer, Growing Agile*

"*More Agile Testing* is an extensive compilation of experiences, stories, and examples from practitioners who work with testing in agile environments around the world. It covers a broad spectrum, from organizational and hiring challenges, test techniques and practices, to automation guidance. The diversity of the content makes it a great cookbook for anyone in software development who is passionate about improving their work and wants to produce quality software."

—*Sigurdur Birgisson, quality assistance engineer, Atlassian*

Chapter 23

TESTING AND DEVOPS

We've worked for many years with team members who do activities that are now called DevOps. As Jez Humble says, team members engaged in DevOps build "a platform that allows developers to self-service environments for testing and production (and deployments to those environments)" (Humble, 2012). They provide tools that enable delivery teams to build, test, deploy, and run their systems. In our experience, everyone on agile teams is doing DevOps activities, but one or more team members contribute in-depth specialized skills. Giving the skill set a label has called attention to its importance in building quality into software. DevOps is an integral component of successful agile testing.

A SHORT INTRODUCTION TO DEVOPS

The term *DevOps* was first popularized by the DevOps Days conference in Belgium in 2009. Since then there have been some excellent publications devoted to DevOps and how it fits into agile development. *Continuous Delivery* (Humble and Farley, 2010) and *DevOps for Developers* (Hüttermann, 2012) are two good guides to DevOps.

DevOps includes practices and patterns that improve collaboration among different roles in delivery teams, streamlining the process of delivering high-quality software. These practices and patterns help teams write testable, maintainable code, package the product, deploy it, and support the deployed code.

One area of focus for DevOps is shortening cycle times. Some businesses are able to take this to an extreme, delivering new code to production

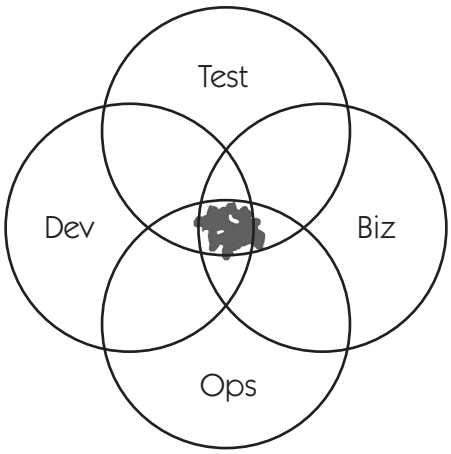


Figure 23-1 Intersection of development, testing, operations, and the business: the whole-team approach!

many times per day. Delivery team members find reliable, repeatable ways to get from an idea to delivered business value, with a cycle time that's not only short, but safe. This helps prevent defects and build in quality.

DevOps activities are designed to improve the maintainability and speed of automated tests and deployment. Examples include maintaining test environments, providing fast feedback from continuous integration (CI), and ensuring reliable and frequent—perhaps even continuous—deployment. Many teams now test their infrastructure continually to guard against spurious intermittent regression test failures due to server, deployment, or configuration failures. Business stakeholders are also involved, helping to hire the right people, obtain the necessary hardware and software, and gather information to guide future development. Figure 23-1 illustrates that DevOps is the intersection of programming, testing, operations, and the business. It is another way to look at the whole-team approach to agile testing.

DevOps and Quality

Agile team members' skills often include scripting, system administration, coding, configuring CI, tooling, and collaboration and communication skills that enable them to perform DevOps activities. These activities are instrumental in agile testing and continually building quality into our products.

Janet's Story

Years ago, before the idea of continuous integration existed (at least in my world), I worked with a person I consider to be one of the first DevOps practitioners. Darcy's role included implementation of our product at client sites, first-level support for clients, keeping our servers running, as well as being our technical tester, setting up our test environments, and keeping them running. He was able to bring what he knew from the world of clients into our testing, including recovery testing. I did not realize the value he brought to the development of our product until I worked in an organization where that crossover role did not exist.

In another organization, we had an infrastructure test team. They worked with the programmers and the teams that supported the hardware and communications. They worked with each of the agile teams to help them understand the impacts of changes to the infrastructure on the new features.

People with specialized operations and system administration skills help the team improve quality on many different fronts. They can help set up appropriate development, test, and staging environments, optimize the CI, and refine the deployment processes for these environments. They can also help find or build and implement automation frameworks and other tools that suit the team's needs. In our experience, they're terrific at diagnosing problems and have many useful tools at their disposal. They can help with the infrastructure to support automated tests, such as generating test data.

Figure 23-2 illustrates the idea that DevOps crosses into all four of the agile testing quadrants. It helps guide development with many different types of testing activities, provides tools and environments for evaluating the product, and builds the technology that enables testing of diverse quality attributes such as performance, reliability, and security.

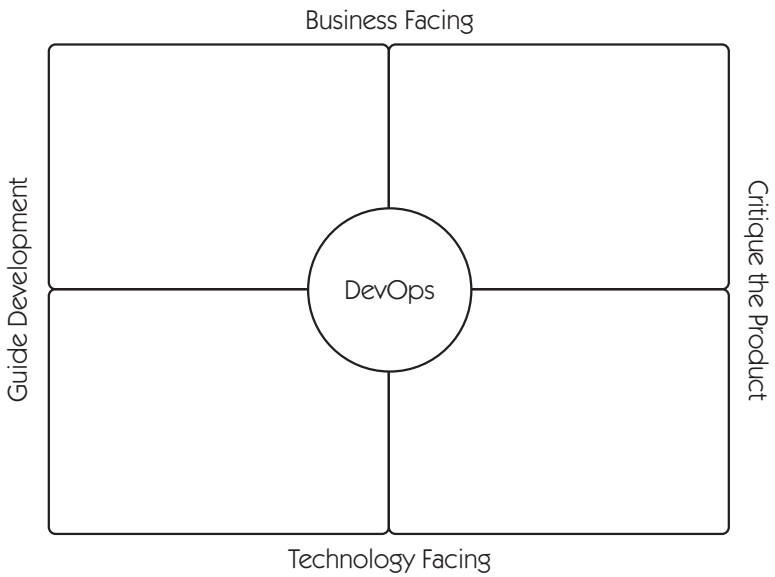


Figure 23-2 Test activities intersect at DevOps.

Lisa's Story

Back in the mainframe days when I worked as a programmer/analyst, I collaborated closely with the machine room operators. Together, we learned ways to solve and prevent problems with processing batch jobs.

In the early 1990s I was fortunate to learn UNIX system administration skills from experts on my team. I learned the importance of repeatable release processes and ways to verify the packaged product. In the 2000s, my teammates with in-depth system administration skills helped find better test automation frameworks and ways to configure CI jobs for quicker test feedback. They showed me how to use monitoring and log files to help with debugging and exploring.

My current team is the first I've worked with that uses the term *DevOps*. We have a backlog of DevOps user stories to improve all aspects of our infrastructure, including our test environments, CI, and deployment process. Interested team members join a weekly DevOps meeting to prioritize work, and these activities are planned along with product features. Programmers work with the company that hosts our production site and maintains our search engine software. We do our own production deploys and can respond quickly to operational problems. We experiment with ways to continually improve our delivery process.

Some companies maintain separate operations or IT departments. The members of these silos often have to support multiple teams, juggling development needs with production maintenance. They may lose touch with the development teams and their needs.

Lisa once facilitated a workshop for a large company where participants from teams working on different product areas identified issues that were impeding various aspects of their testing. Testers complained that the other parts of the system had incompatibilities with theirs, and they had no idea how to overcome these. One of the workshop participants was from the operations department. Once he saw the obstacles that were listed, he said, "I had no idea you were struggling with this. My team can take care of these problems. From now on, contact me directly!" Sometimes solving a thorny testing problem just takes getting people from different departments in the same room.

Seeing the Whole: Add Infrastructure to the Testing Scope, DevOps Style

Michael Hüttermann, *author of* DevOps for Developers (Hüttermann, 2012)*, shares a success story he had with a whole-team approach to supporting infrastructure and testing through DevOps.*

In a bigger project with about 100 developers, we had to cope with aggressive time-to-market targets, protecting competitive advantages, a lot of technical complexity, as well as high demands on availability and capacity. These were the main drivers to implement the DevOps approach. This approach included applying "the infrastructure as code" paradigm, which starts with putting Puppet manifests to version control, in our case Git. [*Authors' note:* See the "Tools" section of the bibliography for links to all tools mentioned in this chapter.]

An end-to-end, department-spanning delivery pipeline was in place, starting with stage 0, the developer workspace, and closing with higher test machines, a production mirror, and production. From the version control system, baselines were created continuously. Those baselines contained different configuration units that made up a release, including business code, unit tests, integration tests, and infrastructure information. Building on top of these baselines, we created release candidates continuously. Cherry-picked release candidates

were promoted to be released to production. Those releases were both fit for purpose (by containing the functional scope) and fit for use (by conforming to the nonfunctional requirements).

After we applied DevOps concepts, the processes and tools of both development and operations were aligned with each other. Each used the same approaches to provision machines, including installation and configuration of infrastructure, middleware, and the business application. Development and operations continually collaborated to ensure maximum knowledge and open information exchange. We had slack time to learn and experiment, and we built mutual respect. We used kanban to manage the flow and reviewed the design early in the process. This enabled early and frequent learning, and "failing fast."

By applying DevOps practices, service-level requirements and service capabilities could be defined and checked early in the process. Developer machines and test machines were production-like, thus they delivered fast and meaningful feedback of installation and configuration, as well as nonfunctional requirements such as security and monitoring.

As part of the continuous delivery platform, Puppet was used together with Vagrant and Jenkins to set up and remove virtual machines for reproducible testing of the provisioning process itself, as well as the defined result on the target machine.

In order to find errors early and often, we started with checking for more basic failure categories. As part of a continuous build, the infrastructure code in the form of Puppet manifests was validated early in the process. When syntactical errors in the manifests were found immediately, the process was aborted and no binaries were produced for further usage. Finding this category of errors is easy to achieve by just applying a `puppet parser validate` as part of a dedicated build step in the continuous build.

After the actual configuration of the target test environment with Puppet, another downstream stage was added to check the correct provisioning. We introduced test manifests that were applied by puppet `apply-noop` to check for Puppet compilation errors, and then checked the resulting log of events. A basic automated smoke test compared actual and desired results.

When the delivery pipeline was mature enough, we added code-style verifications. We found it helpful to agree on a shared format for all our artifacts. Many checks on business code were done with SonarQube, but we also checked style guide compliance of infrastructure code with `puppet-lint`.

Applying DevOps practices taught us that it was crucial to foster the spirit of the "one team" that consists of members of development, coders, testers, business, and operations, along with network, systems, and database engineers. All involved experts became developers of the solution.

The combined team shared business goals such as reducing cycle time. Shared processes, such as using the same provisioning concept for all machines, and shared tools, such as Puppet, also aligned the combined team. Many production incidents are caused by changes to the IT infrastructure or by unplanned work, such as firefighting production incidents due to broken processes or badly tested solutions. Puppet allowed us to define executable specifications of target infrastructure behavior, making the infrastructure testable. This made automation reliable, which shortened feedback cycles. The documentation about the machines is always up-to-date, facilitating conversations between developers and operations staff.

It was elementary to distinguish between accountability and responsibility. Colleagues of the "one team" of devs and ops were collectively responsible, but only one person from the development department was accountable for development machines and one person from the operations department was accountable for operations machines. Other colleagues who were neither responsible nor accountable were consulted or informed, early and often.

Overall it was a great success to emphasize the entire flow of work from start (inception) to end (operation), to extend agile development testing practices to operations, and to intensively include operations staff starting with the early phases of software development.

To achieve the reliable test automation and shortened feedback cycles that Michael's team enjoyed, you need to understand your build pipeline. Know what tests run in which environment. The concept of build pipelines works well for talking about test environments. Building good test environments has been an obstacle to many of the testers and teams with which Janet has worked. In Chapter 5, "Technical Awareness," Figure 5-2 was an example of a very simple build pipeline. Generally speaking, there is an automated push to the development environment if the CI tests all pass. However, one of the practices that we recommend is to use a pull system for the test environment. This gives testers control over their test execution. In our opinion, there is nothing worse than to be halfway through an exploratory test cycle, only to have the build

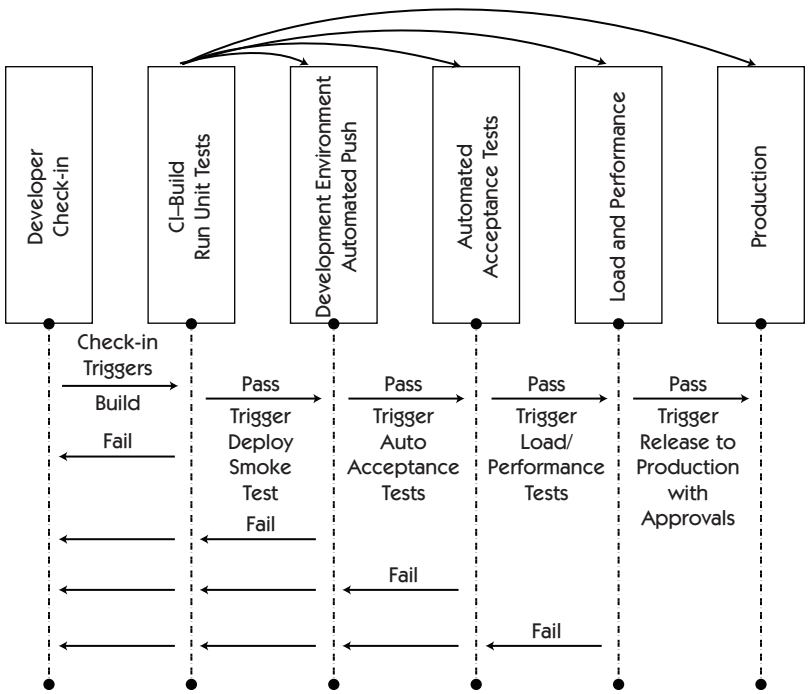


Figure 23-3 Automated build pipeline

you were testing overwritten. That often means starting the testing over from scratch.

Figure 23-3 is a diagram of an automated build pipeline. If you compare it to the simple build pipeline in Figure 5-2, you'll see that the test and the staging environments in this example are not part of the automation. Those are two environments where you would perform exploratory testing.

In Chapter 18, "Agile Testing in the Enterprise," we presented Dell's testing strategy for its enterprise solution. This next story gives more details on the infrastructure needed.

Automated Build Verification Testing

Kareem Fazal, *a software engineer at Dell, talks more about their complex build verification and how they adapted it after their first attempts.*

The Hardware Test-Matrix Challenge

One of the challenges faced in virtually all of our software test environments at Dell Enterprise Solutions Group (ESG) is the extensive hardware compatibility matrix that our software is expected to support. This is most evident in our Server Systems Management firmware and software projects supporting our twelfth-generation servers. When we first applied agile practices to our 12G servers, we established a CI and automated build verification test (aBVT) environment. This CI and aBVT environment supported a project with 15 Scrum teams distributed across India and the United States. Since new code was checked in several times an hour, there was a constant demand for frequent, "known-good" builds. The automated build and tests needed to finish within 90 minutes, running against multiple physical unit-under-test (UUT) environments that were configured with specific network interface card (NIC) and RAID combinations.

The procedures that we initially used for aBVT were not very flexible. It was difficult to add test suites that did not follow a predefined interface format to the existing suite. Also, since test cases were bound to servers, if a specific configuration was required that wasn't already in place, a server had to be taken offline and reconfigured. This meant that we had to be very careful about what tests were run and how much time a test scenario took in order to balance resource use with testing requirements. Under these conditions, it was easier to simply reduce the number of tests run and limit each test run to only those test cases that matched the hardware configuration that was in place. To make matters worse, this was a single-threaded environment, so although there were multiple incoming build requests every hour, they were bottlenecked behind the single, in-progress build verification test run.

The Solution

In preparation for the next update release, the CI and aBVT team took steps to address the limitations of the test environment. The build team envisioned a solution that provided

- Scrum teams with "ease of use" in developing and submitting test cases to achieve maximum hardware coverage
- Build teams with "efficient use of hardware resources" without requiring manual intervention between test runs
- Scrum teams with the "flexibility" to prioritize test cases on specific configurations

In response, our team developed the generic resource manager (GRM). The GRM achieved the flexibility and ease of use we were seeking by having all configurations defined by a simple set of standalone XML files. The name space was defined in a way that did not place arbitrary limitations on what information could be defined, while remaining basically simple and easy to understand. The manner in which the configuration was used allowed for good scalability as well. The tester could define as much or as little information for each testing scenario as was required with very little superfluous effort.

The GRM achieved efficient use of server resources by making sure that they were used in parallel as much as possible, and it allocated resources based on the test requirements. If a test takes longer than expected or ends with an error, the GRM adapts to the conditions at hand. Resource requirements that are defined for a particular test can be as broad or as narrow as required without having an effect on other tests in the test run.

The GRM can be used for small-test setups, but it can also scale and support complex scenarios with many types of resources and thousands of tests in the mix. This is accomplished with a robust, easy-to-understand configuration methodology using a standard XML file format.

Results

With the deployment of the GRM into the aBVT environment, the teams were able to maximize test matrix coverage for each run of a test suite. Because test cases for a particular run did not always require, or consume, all available configurations, additional aBVT runs were initiated in parallel, to make maximum use of available configurations. The GRM provided the build team with the flexibility to scale up and add configurations and test cases without disabling any in-progress test runs, easing maintenance concerns.

As you can see from Kareem's story about his experiences, there is no "one size fits all." Dell ESG custom-built a flexible solution that worked for teams around the globe. Look at your own environments, and strive to simplify the build and deploy and test cycle so that it is consistent and maintainable.

How Testers Add DevOps Value

One area of DevOps where testers can contribute their specialized skills is in helping to test the infrastructure. They can help identify places where infrastructure might break simply by asking questions like, "How do we test what happens if the network goes down?" Testers may not know all the answers, but they are good at asking questions.

Janet worked with a company whose infrastructure testing group worked closely with the individual feature teams, but on a more formal basis. They created an agreement with the teams that looked something like this.

> *The "Levels of Service" that infrastructure testers are prepared to provide are:*
>
> - ***No involvement:*** *If testing is required, it will be planned, executed, and monitored by a representative from a team appropriate to the activity (e.g., DBA, Network, Infrastructure).*
> - ***Guidance:*** *Infrastructure testers will provide guidance to assist with planning an appropriate test. The test will be executed by a representative from a team appropriate to the activity (e.g., DBA, Network, Infrastructure). Testers may review test results and assist with risk assessment for a release/acceptance decision.*
> - ***Planning:*** *Infrastructure testers will prepare a risk-based test plan that defines the testing approach and scope, lists tests to be conducted and acceptance criteria, and identifies risks associated with the project. This plan will be turned over to a representative from a team appropriate to the activity (e.g., DBA, Network, Infrastructure) for execution. Infrastructure testers may review test progress and results and assist with risk assessment for a release/acceptance decision.*
> - ***Infrastructure testing:*** *Testers will prepare a risk-based test plan that defines the testing approach and scope, lists tests to be conducted and acceptance criteria, and identifies risks associated with the project. Infrastructure testers will execute the tests, providing feedback to*

the project team on execution status. Issues will be recorded, tracked, actioned, and escalated using the standard defect-tracking system. Infrastructure testers will present test results and risk assessment for a release/acceptance decision.

Those definitions are more formal than we'd normally use in agile teams. However, it shows how testing specialists can contribute in different situations, helping to manage risk and provide feedback about the infrastructure, including the test and production environments. Earlier in the chapter we learned how Michael Hüttermann's teams tested their infrastructure using open-source tools such as Puppet, Vagrant, and Jenkins. We'll look at more examples.

Testing Infrastructure

Stephan Kämper, *a tester from Germany, explains more details about how he tests infrastructure.*

In my context, the infrastructure is the extra hardware and software that you need to execute tests. It typically includes a CI system, including jobs the system executes; build tools like make, rake, or Maven; and the machines on which the CI runs. Whether or not the (wireless) network or firewall should also be considered part of your testing infrastructure depends on your own context.

In my current team (as of late 2013), we run automated checks against every single commit to the source control system, create new software artifacts (deployable build items) regularly, and run stand-alone integration tests against these artifacts on newly built virtual machines. These artifacts are put through a system integration test and finally brought onto the production system via the CI system.

The whole workflow, from a single commit to production, is orchestrated and supported by the CI system. Since we wrote the code to do all this, it seemed like a very good idea to also test it. The ability to go live with changes depends on that code.

What Part of the Infrastructure to Test

I do not suggest that you test your CI system as such, and neither do I recommend that you test your build system. However, I do think the code you write and feed to these systems should indeed be tested.

As an example, let's assume you're using Jenkins as the CI system and rake as the build tool. We use the Jenkins Job DSL/Plugin, which allows us to write Jenkins jobs as code, in contrast to the usual way of setting them up via the graphical user interface. That way we can have our Jenkins code under version control.

Since rake tasks are Ruby code, the tasks delegate the work to Ruby classes, modules, or methods, which in turn can be developed and tested like any other code. The same is certainly possible for other combinations of build systems and programming languages.

One Caveat

Some Jenkins jobs actually change the "world"—that is, they deploy software to production systems. This is hard to test in a laboratory environment since the whole point of a production system is, well, leaving the safety of a test environment.

But even in this case, you can parameterize the Jenkins jobs so that you can deploy to a test or staging system and production using the same script. That way, deploying to a test system is in fact the test for deploying to production. Note that even when this works nicely, the deployment to production may still be broken, since you might deploy to the wrong server.

Stephan emphasizes the value of testing automated deployments. Other types of infrastructure testing that testers might perform are often part of Quadrant 4 tests, including connectivity, reliability, failover, or backup and restores.

Teams are often called upon to test in production, monitoring site activity or performance. Pairing to monitor production log files has advantages similar to those of pair programming and testing. It is easy to concentrate on one thing and completely miss another, so a second pair of eyes is invaluable.

We've talked about what DevOps does for testing and quality and how testers can help with DevOps activities. In the final part of this section, we'll cover the intersection of both.

Automated Provisioning of Configuration Base States

Ben Frempong, *a storage test engineer, continues the Dell story, describing how they automated configuration base states to make the setup for testing much easier.*

In mid-2012, our Storage Test team was faced with budget and staffing challenges. We had to devise a sustainable solution to a reality that faces most test organizations at one time or another: how to do more with fewer resources while continuously improving the quality of products we deliver to our customers. The solution required us to reinvent and implement sustainable and innovative automation strategies in our testing process to achieve three key goals:

1. Maximize the utilization of our hardware resources.
2. Improve the efficiency of our test execution staff.
3. Capture automation metrics.

The first task was to establish some standards for our automation initiatives. We chose Python as our standard scripting language and selected a centralized, internally developed automation framework for remote script deployment. Next, we focused on two teams that had similar needs: one local and the other remote. After analyzing their daily tasks, we organized their testing activities into the following workflow sequence:

1. Configure base hardware.
2. Install OS.
3. Install updates (firmware/BIOS/OS device drivers).
4. Install solution/product peripherals.
5. Install and configure solution/product-specific packages.
6. Run a suite of manual and automated test cases.
7. Perform exploratory testing.
8. Tear down and reprovision the system for another configuration.

I observed that the first three steps, the base state of provisioning workflows, was often a manual, time-consuming, and repetitive sequence of tasks with the same operating system, on the same hardware or platform families. There were sometimes minor configuration changes such as add-in storage peripherals, but these changes did not impact the base state of any configuration. Additionally, the base state

configurations often had to be redeployed to validate software build releases or bug fixes during a program's test cycles. An automated process could be used to create disk images, files containing the contents and structure of a disk volume or data storage device, with the necessary base configurations. These images could be used to automatically reprovision each configuration's base state.

Existing imaging solutions at that time worked very well in large-deployment environments but not so well in test environments, where one is continuously reprovisioning the same hardware. Also, they took up to 90 minutes or more, depending on the operating system, to capture an image. It might take several hours to restore non-Windows images such as Linux OS. This explained why testers preferred the manual method. We tried to automate existing off-the-shelf imaging solutions, but no single solution supported automated image capture and restores for all three required system environments that we needed to support: Windows, Linux, and ESXi5. Considering the matrix of hardware that needed to be validated, we would have to either purchase more hardware and assign more configurations per tester, or add more testers. Due to our budget challenges, neither solution was an option.

Instead, we prioritized automating the provisioning workflows with some innovative ideas by developing our own fully automated imaging/redeployment solution. First, we repackaged the kernel for the open-source imaging tool Clonezilla to work with our automation framework. Next, we wrote fully automated imaging and redeployment scripts for Windows, Linux, and ESXi5 images. These scripts automatically capture and redeploy base-state images in 10 to 15 minutes or less.

The results have been positive. One of the project teams had typically required up to three testers per test iteration to cover 12 to 18 configurations. After implementing the automated imaging and restore tasks, the team lead was able to single-handedly execute the test iteration himself without additional resources. He even had more time for exploratory testing because the test execution framework allowed him to launch a sequence of automated tasks, which could run overnight or during lunch.

With our new approach to workflow test automation, we are now able to do more with fewer resources, while giving our skilled staff more time for exploratory testing. This allows us to continue to deliver quality products. We are also able to reduce expenditure and maximize the utilization of our hardware resources. Last, from a process perspective, we are able to use the information about our automation capabilities in our test planning, test strategy, and business decisions.

Testers, programmers, and operations experts can collaborate on the CI process to make build jobs more robust, test results more reliable, and deploys to various environments timely and sane. If the whole team understands their operating environments, they may save many hours spent investigating test failures. For example, DevOps may be able to help create checks for connectivity or configuration before deploying, which eliminates wasted time rejecting builds. Alternatively, testers and programmers can analyze test suites to identify duplicate tests, tests that don't add value, and ways to improve reliability. They can also look for gaps in regression test automation and work to fill them.

DevOps practices help our teams efficiently build regression test code and deploy it as needed to environments that support effective exploratory testing. Skilled DevOps practitioners help us implement tools to support testing all necessary quality attributes such as security, performance, and reliability. These abilities are essential for cross-functional teams to deliver high-quality software that provides business value frequently, at a sustainable pace.

SUMMARY

DevOps is a perfect example of the whole-team approach to quality at work. It brings together generalizing specialists with different T-shaped skills to help ensure several aspects of quality. Even when development and operations are separate departments, they can work together to achieve shared goals. Here are some points about DevOps that we covered in this chapter:

- DevOps is a blend of development, testing, and operations engaged in practices that streamline the delivery process, provide timely feedback to improve cycle time and software quality, and facilitate collaboration among all roles on the team.
- Team members with operations and system administration skills collaborate with other team members to build a CI and deployment infrastructure that supports short cycle times.
- Development teams, including testers, can help implement a test-guided approach to building that infrastructure and ensure that the infrastructure continues to operate effectively.

- DevOps practitioners help build and implement test automation drivers, libraries and frameworks, and test data generation tools.
- Understand your team's build pipeline and test environments to take advantage of where you can use automation to supplement your exploratory test efforts.
- Each organization needs to experiment with different hardware configurations to find ones that provide flexibility, consistency, and maintainability.
- Testers can contribute to infrastructure testing by asking good questions and helping with a risk-based approach as well as by actively engaging in testing configuration and deployment.
- Automating the provisioning of configuration base states lets teams do more with fewer resources, using sustainable automation strategies that provide useful metrics and enable more exploratory testing time.
- DevOps can help the team find ways to overcome impediments such as fragile or slow automated tests.