The Highways and Country Roads to Continuous Deployment

Marko Leppänen, Tampere University of Technology

Simo Mäkinen and Max Pagels, University of Helsinki

Veli-Pekka Eloranta, Tampere University of Technology

Juha Itkonen and Mika V. Mäntylä, Aalto University

Tomi Männistö, University of Helsinki

// Interviews with 15 information and communications technology companies revealed the benefits of and obstacles to continuous deployment. Despite understanding the benefits, none of the companies adopted a fully automatic deployment pipeline. //

IN CONTINUOUS INTEGRATION, software building and testing are automated. Continuous deployment takes this practice one step further by also automatically deploying software changes to production.[1] Essentially, continuous deployment gives programmers the power to deploy software at will.

Continuous deployment has several apparent upsides, the primary one being a shorter time to market. In addition, it allows the release (and removal, if necessary) of new features and modifications with as little overhead as possible. This lets companies experimentally determine the value of software components, with real users.[2] However, transitioning to a continuous release cycle is nontrivial and might require significant process changes, even when employing agile methodologies such as Scrum.[3]

Because continuous deployment is relatively new, there have been only a few empirical studies. These studies have focused mainly on conceptualizing the levels of continuous deployment and the challenges of and barriers to achieving it.[4,5] As part of N4S (Need for Speed; www.n4s.fi/en), an industry-led Finnish information and communications technology research program focusing on fast deployment, we set out to gain an in-depth understanding of how continuous deployment works in the field. In particular, we wanted to know the state of the practice, the perceived benefits of continuous deployment, and the obstacles to its adoption.

So, we conducted a qualitative semistructured interview with open questions in 15 companies. Of the 26 industry partners involved in the research program, we selected 12, partly on the basis of their availability. We selected companies from different domains and of varying business sizes to provide a rich understanding and characterization of continuous deployment. We added three companies not involved in the project, to further deepen our understanding in certain domains.

0740-7459/15/$31.00 © 2015 IEEE

The study aimed to be exploratory and provide a deep understanding through detailed analysis of information-rich cases in a realistic context. Our findings describe continuous deployment in practice and are hypothesis generating rather than confirmatory.

For each interview, we asked the companies to refer us to a representative of the development team that was the most advanced in terms of continuous deployment. We intend the gathered data to give grounded examples of what factors to take into account when adopting continuous deployment, not to provide universally generalizable findings. We analyzed the data using thematic analysis and synthesis.[6]

The State of the Practice

The naive way to measure continuous-deployment adoption is to use a simple yes/no scale, where "yes" denotes a fully automatic chain to deployment and "no" a chain with one or more manual steps. However, such a dichotomous scale wouldn't accurately capture situations in which companies have taken steps toward automated deployment without yet realizing a fully automatic pipeline. It wouldn't give the degree of adoption or any other sense of how close a company is to a fully automatic chain.

To avoid this problem, we used five discrete measures to determine the adoption level:

- *The fastest possible time for a code change to propagate to production*. This indicates the speed with which a development team can push a change to production using its normal development workflow. In this case, "normal" excludes "monkey patching" or equivalent situations in which changes are patched into production systems without adequate testing or code review.
- *The cycle time to potentially deployable software*. This indicates how fast increments of software are delivered internally but not deployed to production.
- *Whether the company uses an automatic chain to potentially deployable software*. This indicates whether code integration and testing are automated.
- *The actual cycle time to deployment*. This indicates how long it actually takes to push software changes to production.
- *Whether the company uses an automatic chain to deployment*. This indicates whether code integration, testing, and deployment are automated.

These measures provide an overview of companies' maximum internal delivery capabilities. They also illustrate the situation in real-world projects in which external factors might limit the extent to which companies can achieve an ideal continuous-deployment pipeline to production.

Table 1 presents the degree of continuous-deployment adoption in the companies, along with basic anonymized demographic information. Of the 15 companies, five were small businesses, with fewer than 20 employees. Six were medium-sized, with 80 to 250 employees. The remaining four were large, with thousands to tens of thousands of employees.

Only one company had a fully automatic pipeline to potentially deployable software. No company had an automatic pipeline all the way to deployment in a production environment. The company's size didn't appear to matter, but the operating domain did. For example, companies in the Web domain could deploy significantly more often than telecom companies. We explore the reasons for such discrepancies later.

> No company had an automatic pipeline all the way to deployment in a production environment.

The Perceived Benefits

We wanted to know why companies have transitioned toward more automatic deployment. Empirical studies haven't focused on the benefits of and motivations for continuous deployment, and even the benefits of the more established practice of continuous integration seem unclear.[7]

We asked the interviewees to describe what benefits they thought continuous deployment has. We uncovered the following six categories of benefits.

Faster Feedback

The most frequently mentioned benefit (14 interviewees) was the ability to get fast feedback to development. The feedback flowed both from the development process directly to developers and from customers to the development organization.

TABLE 1

The interviewed companies' continuous-deployment capabilities.*

Company ID	Primary domain	Fastest possible time for a code change to propagate to production	Cycle time to potentially deployable software	Automatic chain to potentially deployable software?	Cycle time to deployment	Automatic chain to deployment?	Size of organizational unit (no. of persons under one continuous integration or continuous deployment)	No. of people in company
A	Medical embedded systems	—	1 mo.	No	1.5 yrs.	No	10	<100
B	Industrial automation	1–2 days	1 hr.	No	—	No	50	15,000
C	Telecom network	2 days	1–2 wks.	No	3 mos.	No	~100	>10,000
D	Telecom network	4 wks.	2 wks.	No	3 mos.	No	~350 (3 sites, 20 teams)	>10,000
E	Web development	30 min.	1 wk.	No	1 wk.	No	7	80
F	Web development	1 day	13 days	No	3–4 mos. or 2 wks., depending on the product	No	8	80
G	Web development	5 min.	20 min.	Yes	1 hr.	No	7	7
H	Web service	1/2 hr.	1 day	No	1 wk.	No	< 10	250
I	Web service	1 hr.†	1 day	No	1 wk.	No	(3)	180
J	Web framework	1 day	2 wks.	No	2 wks.	No	20 (7)	80
K	UI framework	2 days	1 wk.	—	6 mos.	—	200	1,000
L	Mobile games	1 wk.	2–3 wks.	No	1 yr.	No	3 (3)	9
M	Mobile games	1 wk.	—	No	—	No	7	7
N	Mobile games	1–2 wks.	1/2 hr.	No	—	No	9 (4)	9
O	Mobile applications	2 hrs.	2 days	No	1 wk.	No	17 (10)	17

** A blank cell indicates insufficient interview data.*
† 3.5 hrs. for full tests.

Interviewees described feedback as providing developers quick, clear visibility of what features have been completed. The immediate feedback through continuous integration minimized, for example, the need for developers to wait for test results before proceeding with their activities. Interviewees perceived that such immediate feedback reinforced developers' sense of accomplishment and heightened their motivation.

Another benefit was the ability to receive more frequent feedback from customers and users. The interviewees perceived that continuous deployment gave developers deeper insight into which features were really needed and whether implemented features worked for customers. Continuous deployment made it possible to experiment by getting instant feedback from end users on alternative solutions. So, decisions on what features to continue developing or to discontinue were easier.

More Frequent Releases

Six interviewees saw the higher frequency of releases as an advantage. They emphasized the radically decreased time-to-market after transforming to a continuous process. The interviewee from company D, a large development organization, reported improvements in delivery capability "from six months to two weeks," even though the actual deployment didn't follow that pace.

Interviewees also associated more frequent releases with less waste because the features weren't waiting in the development pipeline to be released. They felt that more frequent releases provided value to customers by showing tangible results sooner. This agility also enabled stakeholders to stay informed of the development status.

Improved Quality and Productivity

Continuous deployment emphasizes build and test automation together with a much reduced scope for each release. Seven interviewees felt this helped improve code quality and application functionality. Also, truly robust automated deployment entails having a comprehensive test suite, which in turn will likely improve quality.

Improved Customer Satisfaction

Five interviewees felt that customer satisfaction improved because the developers could respond more quickly to customer feedback in terms of new features and maintenance releases. These interviewees perceived that the shorter lead time of new product features provided better customer service.

Continuous deployment emphasizes build and test automation together with a much reduced scope for each release.

Effort Savings

Reduced development effort wasn't central in the perceived benefits. However, three interviewees reported that a more continuous process helped streamline the release process and eliminate manual work. Automating tasks that were previously manual saved time and thus effort—the magnitude of which depended on the amount of manual work before automation.

A Closer Connection between Development and Operations

One interviewee brought up the closer connection between the development and operations people. Continuous deployment prevents development silos, where the development occurs over long periods without connection to the production environment. This is because releases must be communicated to and coordinated with operations teams on a tighter schedule than before.

Discussion

Faster feedback, improved quality and productivity, and decreased effort could be attributed to both continuous integration and continuous deployment and were similar to the reported benefits of continuous integration. The benefits that seemed unique to continuous deployment were more frequent releases, improved customer satisfaction, and the closer connection between development and operations. Rally Software reported similar motivations for continuous-deployment transformation, along with improved productivity and quality.[3]

Goals for Continuous Deployment

The interviewees didn't necessarily view the ultimate goal of fully automated deployment to production as the best deployment model. So, it's important to understand what practitioners in different contexts feel is the desirable level of continuous deployment. We asked the interviewees to describe their company's goal for continuous deployment. We identified the following four main categories (company E had case-dependent goals, so we didn't categorize it).

Fully Automated Continuous Deployment

Companies G, K, and I emphasized removing all manual testing and other obstacles to full continuous deployment—such as legacy code—from the release process. The goal was clearly to achieve fully

automated deployment, at least through some staging environments before production.

Continuous-Deployment Capability

Companies B, C, and D aimed for the capability to continuously deliver from the development organization, but not directly to production. These companies were developing large telecom systems or industrial automation and didn't see the need for full continuous deployment in their domain, owing to regulations in the field and customer preferences.

On-Demand Deployment

For companies L, M, N, and F, continuous deployment to customers wasn't important or even a desired release model. These companies wanted more control of the release contents and schedules. However, they too strived for fast, automated deployment, only they needed to control the releases' timing. We identified two subtypes of on-demand deployment.

First, companies L, M, and N, which produce mobile games, had a staged release model; they released prereleases to a limited audience to get feedback. A new game's release cycle is long, and these companies felt no need for a more continuous deployment model. Instead, they wanted efficient, automated prereleases. Their goal was to always have handy a potentially shippable product they could use to collect feedback from investors, beta testers, and other early adopters.

The second type of on-demand model involved company F, which took into account customers' needs when planning when and how often to deploy new releases.

Calendar-Based Deployment

Companies A, H, J, and O applied a calendar-based release schedule, which they felt was enough for their needs. These companies had a one- to two-week deployment cycle; two had an internal delivery rate of only a few days. However, their customers were happy with receiving new releases only once or twice a month. These companies felt that rapid updating would be disruptive for end users.

Discussion

Often, researchers see continuous deployment as a universal goal, and they focus on removing the barriers to its full adoption.[4] However, we found that, in practice, software organizations settled on a less continuous process for varying reasons. These context-specific motivations deserve more research and experience sharing so that we can better understand whether there are different types of benefits that could be achieved with varying levels of continuous deployment. For example, companies E, L, and M, which had the potential to deploy once a week or once every two weeks, had no plans to improve the deployment pace because the weekly pace was appropriate for their product and business.

> Adopting continuous-deployment practices involves coordination and work from teams throughout the organization.

Obstacles

Although adopting continuous deployment could provide a number of benefits, the companies weren't rushing to fully streamline the last mile of software delivery to their customers. Obstacles to continuous deployment might manifest themselves as manual labor, but that isn't the only reason why the companies hadn't changed how they work.

We asked our interviewees to describe the obstacles to the full adoption of continuous deployment. We identified eight key themes.

Resistance to Change

Adopting continuous-deployment practices involves coordination and work from teams throughout the organization.[5] Gerry Claps and his colleagues argued that individuals couldn't be the sole agents of change themselves and that extensive support from the superiors was a prerequisite for adoption.[5]

We also found signs of resistance to change. One interviewee mentioned that the company's management downright resisted changes to the current development practices and was unwilling to switch to continuous deployment. Several other interviewees also said they felt their organizational culture wasn't particularly receptive toward new ideas and continuous deployment, which was seen as a challenge.

Social relations and culture naturally vary among companies. Still, it's good to remember that adopting continuous deployment has both social and technical implications and that carrying out the transition requires more than one person.

Customer Preferences

Some customers are perfectly content with software that's updated every three to four months; they might even be reluctant to deal with more frequent releases. Researchers have highlighted the role of customers as a social challenge for continuous deployment;[5] our interview results support this. An interviewee from a telecom provider mentioned that the company tried to ramp up its release frequency from quarterly releases but that the receiving end wasn't prepared to handle a shorter release cycle. Eventually, dissuaded by the experience, the company reverted to its previous schedule.

Domain Constraints

As Table 1 shows, the domain in which a company operated affected the flow of continuous deployment. For instance, for the telecom industry, the domain imposed restrictions on delivery and deployment speed.

According to our interviews, the telecom sector is also struggling with the diversity of its clients. Deploying software releases directly to network devices is challenging because network configurations might vary among clients.

Domain-imposed restrictions also apply to medical embedded systems. The interviewee from company A pointed out that the company emphasized compliance to national or international medical standards. The company needed to ensure that changes had no adverse side effects, guaranteeing patient and user safety.

Software for operating control systems in factories might face almost insurmountable obstacles to continuous deployment. Updating an automation control system with new software can require the factory to stop production for the update's duration. The interviewee from company B, which works on automation control systems, reported that the company might have to stop the whole process for a day or so or run the control system updates during weekends. Such a setting prohibits instant deployment because each update must be scheduled. Also, the costs related to factory downtime can make automation software providers think twice before pushing a new release to production systems.

The distribution channels that provide software to customers might also slow down deployment. When a software development company doesn't fully control the distribution channel, a third party is responsible—at least partly—for making a product available.

A submission to a software market such as an online application store can trigger lengthy external quality assurance processes. The interviewee from company L, which develops mobile games, noted that its company's releases couldn't be instant because an application store took a week to review the submissions and publish the product.

Developer Trust and Confidence

Continuous deployment is demanding for development organizations. Developers must have sufficient proficiency and knowledge of typical continuous-deployment practices. Code going straight to production must be as defect-free as possible. A decent amount of automated testing can somewhat guarantee software quality.

According to the interviews, many companies saw the need to improve their automated testing; they viewed the existence and volume of such testing as paramount for continuous deployment. The lack of automated testing was a major barrier the companies faced on their path to more continuous software releases. Trust in defect-free builds was eroded by automated tests that had relatively low coverage, thus necessitating exploratory testing. Some companies, especially those working with the quirks of Web browsers or mobile games, found automating user interface tests particularly challenging.

> The domain in which a company operated affected the flow of continuous deployment.

Also, developers' reputations are on the line: deploying a broken build to customers could strain the relationship between parties and create an unwanted user experience. Any lack of confidence in an application's quality is amplified by the knowledge that any and all changes are immediately deployed.

In addition, setting up the infrastructure required for continuous deployment could be cumbersome. Knowledge of the continuous-deployment pipeline with its continuous integration, automated testing, and release deployment practices is a prerequisite for developers. However, constructing the pipeline required resources that the companies didn't necessarily have. The interviewee from company G, which

develops Web-based products, noted that the company simply didn't have the time to update the existing setup to better support continuous deployment. In cases such as this, developing software trumps infrastructure setup and configuration.

Legacy Code Considerations

Integration failures due to insufficient testing inhibit the continuous-deployment flow. Untested code partially shifts the responsibility of testing to other developers or quality assurance personnel, potentially delaying the detection of defects until the code is truly integrated. This is the case with many legacy software systems that have been built over decades and might not be designed to be automatically tested. The software quality gradually decreases.[8]

> Integration failures due to insufficient testing inhibit the continuous-deployment flow.

Although legacy code wasn't the most common barrier, the interviewee from company I, which develops Web products, mentioned that it was a challenge for one project. The company couldn't employ fully continuous deployment because of legacy system integration. It's possible to automatically generate a proper test harness for large legacy systems[8] and write additional tests for the largest, recent, and most fixed source code files.[9] However, such a situation might overwhelm legacy system developers.

Duration, Size, and Structure

Writing good automated tests that cover much of the software code takes considerable time for developers. So does running and executing automated test cases. The sheer number of automated tests and the duration of their execution limit delivery speed. Company K, a large company developing a Web development framework, had to execute more than 60,000 automated tests.

Such a huge number of tests inevitably take time to execute. The interviewee from company J, which develops Web frameworks, mentioned that the full test suite took a good hour and a half to finish. Under these circumstances, instant releases are harder to achieve because the release process takes at least as long as test execution, given that all the tests run after the code changes are committed.

The code base's size also affects the time to deploy software releases. Company K, which works with a user interface framework, had to compile, build, and assemble deployable packages out of 8 Gbytes of source code. The interviewees from companies A and F also identified the code base's size and complexity as challenges to continuous deployment.

The interviewee from company F, which develops a back end for Web applications that integrates many data sources, explained that piecing together a release took a full day. One project he was working on had been split to subprojects, each of which had to be compiled and built separately before a release could occur. In this project, the interrelations between the subprojects weren't restricted to building. Development tasks could have dependencies from one subproject to another, and parts of the project could be at different development stages. Because of the project's structure and modularity, achieving continuous deployment would have been difficult.

Different Development and Production Environments

One source of grief in software development and deployment is the different environments used in development, testing, and production systems.[1] Keeping all the environments similar enough and maintaining good configuration conventions can be challenging. Company F's interviewee commented that the company did extra checks because the production system used a different database than was used in development. Also, the code could work differently in the production system, causing unforeseen defects.

Company E's developers were also aware of the troubles different environments might cause. When that company's interviewee started working on a new customer project, he first harmonized the development and production environments so that the developers had a similar production-like environment available on their development workstations through virtualization technologies.

Manual and Nonfunctional Testing

Automated tests that execute after each change don't necessarily cover all essential aspects, especially those related to quality attributes such as performance and security. So, some

companies tended to run a series of additional, partly manual tests before each release to ensure that performance and security requirements are met.

For network infrastructure services, load testing can be resource intensive and requires plenty of hardware, as the interviewee from company C, a large telecom provider, noted. That interviewee also mentioned that creating static test fixtures—which could be used in automated tests—from network data was difficult in this case. This is because the network was constantly evolving and the static data soon wouldn't match the real world.

In company E, which performs Web development, performance testing meant measuring execution times and comparing the results to previous measurements to ensure that performance hadn't degraded because of the changes. According to that company's interviewee, this case required a separate performance-testing phase because comparing the actual production performance of a developer workstation to a hosted Web server could be tricky.

Every manual test takes development further from continuous deployment because releases require more human intervention. Several interviewees mentioned that, besides the possible performance and security tests, they performed a round of manual or exploratory testing, trying out the product on actual devices with or without a test plan.

This approach is certainly understandable in a field such as games, in which the user experience is particularly important and the product should function on multiple devices and platforms. The interviewee from company L, which develops games, stated that the multitude of devices and fragmented base of hardware

ABOUT THE AUTHORS

MARKO LEPPÄNEN is a doctoral student in the Tampere University of Technology's Department of Pervasive Computing. His research focuses on software architectures, agile methodologies, and continuous delivery. Leppänen received an MSc in electrical engineering from the Tampere University of Technology. Contact him at marko.leppanen@tut.fi.

SIMO MÄKINEN is a doctoral student in the University of Helsinki's Department of Computer Science. His research interests include software development methodologies, software quality assurance practices and tools, and software architectures. Mäkinen received an MSc in computer science from the University of Helsinki. Contact him at simo.v.makinen@helsinki.fi.

MAX PAGELS is a software developer at SC5, an e-commerce and customer service solutions company. His interests include data analysis, Web technologies, software measurement, and software development processes. Pagels received an MSc in computer science from the University of Helsinki. Contact him at max.pagels@alumni.helsinki.fi.

VELI-PEKKA ELORANTA is a software developer at Vincit Oy. His research interests include software architectures, startups, agile methods, and continuous delivery. Eloranta received an MSc in software engineering from the Tampere University of Technology. He has been active in the pattern community, has served on the program committees of several PLoP (Pattern Languages of Programs) conferences, and is the program chair of EuroPLoP 2015. He also coauthored a pattern book on patterns for distributed control systems in 2014. Contact him at veli-pekka.eloranta@iki.fi.

JUHA ITKONEN is a postdoctoral researcher in Aalto University's Department of Computer Science and Engineering. His research interests focus on experience-based and exploratory software testing and quality building in varying agile contexts. Itkonen received a DSc in software engineering from Aalto University. Contact him at juha.itkonen@aalto.fi.

MIKA V. MÄNTYLÄ is a professor of software engineering at the University of Oulu. He previously was an assistant professor in Aalto University's Department of Computer Science and Engineering. His research interests include empirical software engineering, software testing, and defect data. Mäntylä received a DSc in software engineering from the Helsinki University of Technology (now called Aalto University). Contact him at mika.mantyla@aalto.fi.

TOMI MÄNNISTÖ is a professor of software engineering in the University of Helsinki's Department of Computer Science. His research interests include software architectures; variability modeling, management, and evolution; configuration knowledge; and flexible requirements engineering. Männistö received a PhD in computer science from the Helsinki University of Technology (now called Aalto University). He's a member of the IFIP TC2 Working Group 2.10 Software Architecture, IEEE Computer Society, and ACM. Contact him at tomi.mannisto@cs.helsinki.fi.

partly stopped the company from doing continuous releases.

The need to support multiple platforms or devices exists in other domains, too. Company E's interviewee explained that the company's product had a public application interface and was used on various devices. Although the release frequency was rapid with weekly releases, the company had to alert a third-party organization a day before the release date. That organization then tested the product on different devices before release. The company couldn't skip this phase because exploratory testing could reveal defects that automated tests didn't catch.

The interviewee from company F, which develops Web products, summarized the situation, saying that the day he no longer found issues or defects doing exploratory testing before a release would be the day his company could start thinking of going to a more continuous release model. That day hadn't yet arrived.

Our interviews give a sound characterization of continuous deployment and its adoption in 15 Finnish companies. Although we now understand continuous deployment's perceived benefits pretty well, they weren't entirely what we anticipated. Faster feedback and more frequent release trumped effort savings, indicating that for developers and team leads, customer satisfaction takes clear priority over development cost.

Also, although these companies fell short of full continuous deployment, they all had the internal capability to release software increments to customers more quickly than they did. External constraints such as domain-imposed restrictions and customers not wanting a faster release schedule hindered the speed of deployment in real-world projects. In the latter case, this was at odds with the perceived benefit of improved customer satisfaction. This indicates the need to deploy frequently enough to satisfy customers but not so much that they become irritated. Internal constraints such as resistance to change and lack of confidence indicate the need to educate and train developers and management.

Several of the companies consciously chose not to fully adopt continuous deployment. Automatic software deployment comes at the cost of relinquishing a certain level of control and human decision making, which some companies might find difficult. Sticking to a familiar, tried, and tested release cycle might be more attractive than sailing in uncharted waters. This is especially true if the company believes that continuous deployment's benefits aren't strong enough to warrant significantly changing its development practices.

If you're looking to employ continuous deployment, you should identify whether any of the discussed obstacles apply in your organization. Doing a thorough job of this will ensure you can choose your adoption goals on the basis of the knowledge of which obstacles can be overcome and which are unresolvable. You can use this in turn as a blueprint to systematically implement changes toward more continuous software development.

Acknowledgments

Tekes (the Finnish Funding Agency for Innovation) supported this article as part of the N4S (Need for Speed) Program of DIGILE (the Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT and digital business).

References

1. J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, 2010.
2. T. Fitz, "Continuous Deployment at IMVU: Doing the Impossible Fifty Times a Day," blog, 10 Feb. 2009; http://timothyfitz.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day.
3. S. Neely and S. Stolt, "Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy)," *Proc. 2013 Agile Conf.* (Agile 13), 2013, pp. 121–128.
4. H.H. Olsson, H. Alahyari, and J. Bosch, "Climbing the 'Stairway to Heaven'—a Multiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software," *Proc. 2012 38th EUROMICRO Conf. Software Eng. and Advanced Applications* (SEAA 12), 2012, pp. 392–399.
5. G. Claps, R. Berntsson, and A. Aurum, "On the Journey to Continuous Deployment: Technical and Social Challenges along the Way," *Information and Software Technology*, Jan. 2015, pp. 21–31.
6. D.S. Cruzes and T. Dybå, "Recommended Steps for Thematic Synthesis in Software Engineering," *Proc. 2011 Int'l Symp. Empirical Software Eng. and Measurement* (ESEM 11), 2011, pp. 275–284.
7. D. Ståhl and J. Bosch, "Experienced Benefits of Continuous Integration in Industry Software Product Development: A Case Study," *Proc. 12th IASTED Int'l Conf. Software Eng.* (SE 13), 2013, pp. 736–743.
8. V. Shah and A. Nies, "Agile with Fragile Large Legacy Applications," *Proc. 2008 Agile Conf.*, 2008, pp. 490–495.
9. E. Shihab et al., "Prioritizing Unit Test Creation for Test-Driven Maintenance of Legacy Systems," *Proc. 10th Int'l Conf. Quality Software* (QSIC 10), 2010, pp. 132–141.