The Top 10 Adages in Continuous Deployment

Chris Parnin, North Carolina State University
Eric Helms, Red Hat Software
Chris Atlee, Mozilla
Harley Boughton, IBM
Mark Ghattas, Cisco Systems
Andy Glover, Netflix
James Holman, SAS
John Micco, Google
Brendan Murphy, Microsoft
Tony Savor, Facebook
Michael Stumm, University of Toronto
Shari Whitaker, LexisNexis
Laurie Williams, North Carolina State University

// On the basis of discussions at the Continuous Deployment Summit, researchers derived 10 adages about continuous-deployment practices. These adages represent a working set of approaches and beliefs that guide current practice and establish a tangible target for empirical validation. //

FACEBOOK HAS SEEN the number of its developers increase by a factor of 20 over a six-year period, while the code base size has increased by a factor of 50.[1] However, instead of slowing down, developer productivity has remained constant as measured in lines per developer. Facebook attributes much of this success to its continuous-deployment practices.

Continuous deployment involves automatically testing incremental software changes and frequently deploying them to production environments. With it, developers' changes can reach customers in days or even hours. Such ultrafast changes have fundamentally shifted much of the software engineering landscape, with a wide-ranging impact on organizations' culture, skills, and practices.

To study this fundamental shift, researchers facilitated a one-day Continuous Deployment Summit on the Facebook campus in July 2015. The summit aimed to share best practices and challenges in transitioning to continuous deployment. It was attended by one representative each from Cisco, Facebook, Google, IBM, LexisNexis, Microsoft, Mozilla, Netflix, Red Hat, and SAS. These 10 companies represent a spectrum from continuous-deployment pioneers with mature implementations to companies with architectural baggage necessitating a multiyear transition to continuous deployment. Deployments of their products range from 1,000 times daily to once or twice yearly. However, all the companies strive to leverage faster deployment to deliver higher-quality products to their customers ever faster. To do this, they use advanced analytics to translate a deluge of available telemetry data into improved products.

Here, we discuss the summit, focusing on the top 10 adages that

0740-7459/16/$33.00 © 2016 IEEE

TABLE 1

A continuous-deployment glossary.

Term	Definition
Branching or branch deployments	A practice in which deployed changes are developed, tested, deployed, and maintained on a branch separate from the main truck of development.
Canary releasing or gradual rollouts	A practice in which code under test is first released to a small batch of real users. If the metrics deviate from nominal ranges, routing to canary release might automatically halt.
Change ownership	A practice in which developers are responsible for software changes for all phases, including development, testing, deployment, and fixing problems.
Configuration management	A process in which an inventory of software and production assets is provisioned and controlled through package managers and tools such as Ansible.
Continuous deployment	A process in which incremental software changes are automatically tested, vetted, and deployed to production environments.
Dark launching	A practice in which code is incrementally deployed into production but remains invisible to users.
Deflighting or rollback	A method for rolling back or decommissioning a defective change and removing it from the deployment pipeline.
Deployment pipeline or automated deployment	A conceptual tool chain or practice for managing the testing and analysis of software and its release to production environments.
Deployment strategy	A method for updating running infrastructure with new versions of software and handling issues such as migrating data, services, and client requests.
End-user communication	A practice enabling communication with users to receive feedback and gather requirements.
Feature flags	A mechanism for dynamically enabling features during production, often controlled by a global in-memory store and cached locally in service instances.
Microservices	An architectural style in which services are created as small and often stateless instances and connected through a central discovery service and property store.
Retrospectives	A practice in which team members discuss the causes and consequences of an unexpected operation outage or deployment failure.
Staging or baking	A stage in the deployment pipeline in which developers test a new software version in a production-like environment. For example, some companies might bake a new version for eight hours before deployment.
Telemetry	A practice in which code is instrumented to compute metrics about feature use and software performance and stability.

emerged from it. These adages represent a working set of approaches and beliefs that guide current practice and establish a tangible target for empirical validation by the research community.

Practices Used at the Companies

Before the summit, 17 teams from nine of the 10 companies completed a survey on continuous-deployment practices (https://github.com/alt-code/Research/blob/master/Continuous/Summit2015.md). The respondents indicated how often their company used each of 11 common practices. (Table 1 defines some of these practices and other continuous-deployment terms.) Although the respondents could indicate partial use of a practice, none did. They either used a practice all the time, didn't use it all, or weren't sure.

Figure 1 summarizes the companies' practices. The most frequent practices were automated unit testing, staging, and branching. The companies also often used code review as a manual signoff in an otherwise highly automated deployment process. We've observed a resurgence of code review, now often handled through lightweight distributed tools, because engineers are more motivated to have others

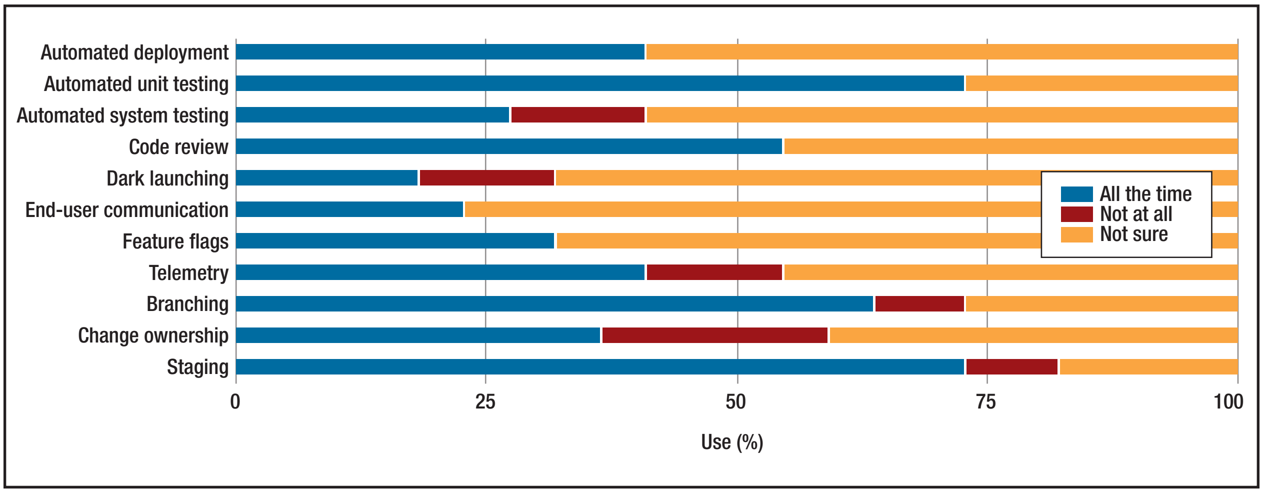


FIGURE 1. The survey respondents' use of 11 continuous-deployment practices (17 teams from nine companies responded). For explanations of some of these practices, see Table 1.

view their code before it's rapidly deployed and defects become public. In addition, the companies used change ownership, in which engineers are on call to deal with the implications of their own defects, rather than the company having a separate field support group that bears the brunt of all defects. So, engineers are more motivated to deploy high-quality software.[2]

The respondents also reported the benefits they realized from continuous-deployment practices. The most prevalent benefits were improved speed of feature delivery, quality, and customer satisfaction. Also, employees were happier because of quicker customer feedback and reduced stress. Although making defects public with rapid deployment might increase stress, the tradeoff is that the stress of missing a release deadline diminishes when the next release train soon leaves the station. In contrast, keeping strict, infrequent deployment deadlines can harm quality.[3] With continuous deployment, management felt decisions were more data-driven with rapid feedback. Teams also believed they achieved higher productivity and better overall collaboration.

In addition, the respondents reported on continuous deployment's challenges. Architecture, safety, and consistency can suffer when development emphasizes delivery speed.[4] With more frequent deployments, the ability to test multiple software configurations is often limited, leaving some common features such as accessibility untested.[5] Teams might resist changes to their development process, especially when traditional roles must be blended into one team. Products with monolithic architectures, technical debt, and few automated tests might have a slower increase in deployment frequency, potentially taking years to reach continuous deployment. Finally, products requiring high levels of safety and regulation might not be able to fully adopt continuous deployment.

The Adages

Although none of the following adages applied to all 10 companies, all the participants agreed with these concepts.

1. Every Feature Is an Experiment

Jez Humble argues that a key to running a lean enterprise is to "take an experimental approach to product development."[6] In this view, no feature will likely persist for long without data justifying its existence.

Previously, feature choices were carefully considered and traded off. Those chosen were designed, built, and then delivered. Evidence rarely supported decisions.

With continuous deployment, developers treat every planned feature as an experiment, allowing some deployed features to die. For example, on Netflix.com, if not enough people hover over a new element, a new experiment might move the element to a new location on the screen. If all experiments show a lack of interest, the new feature is deleted.

Summit participants reported using several supporting practices. Generally, the companies collect statistics on every aspect of the software. They record performance and stability metrics, such as page-rendering times, database column accesses, exceptions and error codes, response times, and API method call rates. For companies to collect this information, the software's architecture must be designed with a telemetry-first mind-set. Instead of keeping localized copies of performance and error logs on a server, the companies stream metrics to a centralized in-memory data store. Finally, to support data analytics, several companies employ a large staff of data scientists, reaching as much as a third of the engineering staff. These companies create and use a rich set of data exploration tools, including custom data query languages.

However, several challenges exist. For example, some companies quickly outgrew the infrastructure for storing metric-related data. Netflix initially collected 1.2 million metrics related to its streaming services, but that soon ballooned to 1 billion metrics. Not only could the in-memory data store no longer keep up, but the company also had to more carefully consider what data was essential for experimentation.

Additionally, engineering queries to extract relevant information for a feature is complex. One participant remarked, "You need a PhD to write a data analysis query." Significant investment in both telemetry and analytics is needed. Investing in these efforts separately can be costly. The participants discussed situations in which they had collected enormous amounts of data but had to redo the experiment because one essential data point was missing.

Nevertheless, not every feature warrants full experimentation, especially non-user-facing features such as those related to storage. Additionally, developers must carefully consider the privacy implications of data collection.

As companies move forward, they'll face the challenge of how to establish a culture of feature experimentation. How can they enable teams to consistently collect targeted information throughout a feature's life cycle without introducing too much overhead or process?

2. The Cost of Change Is Dead

The cost to change code during production can be surprisingly cheap. This contrasts starkly with the predictions that fixes in deployed software would become exponentially more expensive. In 1981, Barry Boehm showed that the cost of change increases tenfold with each development phase.[7] For example, if fixing a change during coding costs $100, fixing it during testing will cost $1,000, and fixing it during production will cost $10,000.

With continuous deployment, the time between development and defect discovery during production is typically short, on the order of hours or days. For example, a developer pushes a new feature into production after two days' work. The next day, a user reports a defect. The fix should be efficient because the developer just finished and can remember what he or she just did. With continuous deployment, all development phases happen the "same day" by the same person or persons, and the exponential cost increase doesn't happen. So, the cost-of-change curve flattens. Thus, a change that costs $100 to fix during development will also cost $100 during production.

Google has found that the scope of changes to review during troubleshooting is small, which makes pinpointing culprits easier and quicker. Also, when changes are deployed into the production environment, the development team becomes aware of release process challenges more quickly through feedback. At Facebook, developers must confirm through their in-house chat system that they're on standby or that their change won't go live during one of two daily production rollouts. So, all developers with outgoing changes can react to any bugs found minutes after going live. Hardly any of the summit participants discussed the cost of changes, indicating that the effects are minimal compared to other cost concerns.

With more traditional release and deployment models, code undergoes rounds of quality assurance to flush out defects. If the release cycle is three to six months, newly found defects might not be addressed for users for another three to six months. Even shorter maintenance cycles are still orders of magnitude longer than daily deployments. Continuous deployment lets developers speedily deploy new features and defect fixes.

Continuous deployment doesn't guarantee that a defect will be found immediately. If it's found later, the cost of change is the same as before. However, if a defect isn't found for a long time, it's likely to be in a low-use feature.

3. Be Fast to Deploy but Slow (or Slower) to Release

Deploying code into production doesn't necessarily mean user-facing features are available to customers right away. Sometimes, a new feature might be deployed and evaluated during production for several months before being publicly released.

For example, at Instagram (a Facebook company), an engineer might want to build a new feature for threading messages on picture comments. By deploying code into production, the engineer can evaluate and test the feature in a live environment by running the code but keeping the results invisible to users by not enabling the new feature in the user interface. This *dark launch* lets the engineer slowly deploy and stabilize small chunks directly during production without impacting the user experience. After stabilization, the engineer can turn on the feature and release it.

Summit participants described several techniques and reasons for slowing down releases. Instagram often uses dark launches to deploy and stabilize features for up to six months before officially releasing them. Microsoft often deploys large architectural changes, using a combination of dark launches and *feature flags*. With a feature flag, a feature is deployed but disabled until it's ready for release; the developer turns the feature off and on through a configuration server. This practice lets Microsoft avoid dealing with integration issues or maintaining long-running feature branches. Deploying changes early, often, and frequently during production reduces the overall deployment friction.

However, this approach poses many challenges. Dynamic configuration lets developers quickly react to problems by disabling features, but developers can just as easily cause outages by inadvertently entering invalid configuration states. Many summit participants reported that although code changes went through rigorous testing and analysis, sufficient tooling wasn't necessarily available to test and evaluate configuration changes with the same rigor. Cleaning up and removing unneeded feature flags is a highly variable practice that often contributed to technical debt. For some summit companies, creating a duplicate production environment, or *shadow infrastructure*, is too expensive or complicated. They're forced to do testing during production, even if that's not strictly desired.

Many techniques can control the speed at which customers see new changes. A company can release software slowly while still deploying every day. Companies must spend extra engineering effort to ensure that delayed-release strategies and testing during production don't negatively affect the user experience.

4. Invest for Survival

Survival in today's market means investing in tooling and automation. Practices once seen as best practices or measures of maturity are now the backbone of a process that relies on rapid deployment. Automated system testing used to be a way to run large test suites to verify that enterprise applications hadn't regressed between releases. Now, these tests are necessary so that developers can get quick feedback and so that releases can be automated to accept or fail a patch. This tooling lets small teams manage large infrastructures.

Companies at the top of the continuous-deployment spectrum, such as Instagram and Netflix, say that tooling pays massive dividends. Facebook found that a small team that's focused on tooling and release automation can empower a much larger team of feature-focused developers. Instagram uses automation to enforce process. Tooling investment allows capturing common workflows and tasks into repeatable, runnable operations that developers or automated systems can perform. Capturing process in tools allows processes to be tested, versioned, and vetted like production code.

Instagram faced challenges with partial automation of a process, which has the risk of developers being unaware of implicit steps needed during deployment. For example, a developer might forget to manually obtain an operation lock on a service (through another tool) before running a deployment command.

Practitioners are seeing that for them to stay competitive and survive, best practices such as automated unit testing are a must. Providing a superior product is now coupled to the speed at which enhancements are deployed. This change requires companies to invest strategically in automation as the scope and scale change over time.

5. You Are the Support Person

Developers have the power and freedom to deploy changes at their own behest. With great power comes great responsibility. If code breaks during production, whose responsibility is it—the developer's or operations team's?

Traditional software methods encourage responsibility silos. Developers "throw code over the wall" to quality assurance (QA) teams, who then throw it over another wall to operations teams. Several summit participants discussed developers who code but don't stop to understand requirements, user stories, or production environments. By owning a feature or code change from cradle to grave (from inception to deployment), the burden is on the developer. This burden means that when things break, the developer is the one who gets the support call and must fix the issue, no matter what time of day.

Because developers own changes from cradle to grave, traditional team structures must change. Netflix has no dedicated operations teams. Although functional roles still exist, such as QA or operations, they're embedded in development teams, creating hundreds of loosely coupled but highly aligned teams. Instead of having a dedicated function (for example, QA, operations, or development), teams have a representative cross section of the necessary roles. Instagram has found value in teams with members who focus on areas but are, as part of the team, ultimately responsible for the life of a feature. Both Instagram and Red Hat have employed support rotations in which each team member spends time handling customer support, which results in shared pain.

Giving teams autonomy comes with challenges—for example, how do autonomous teams integrate with each other reliably? Netflix achieves this integration through a microservices architecture that requires teams to build APIs that they maintain and ensure are stable from change to change. Google enforces team service communication through a common API type and a defined data type that all services must use. With defined communication standards, teams are free to build what they need to accomplish their tasks, in whatever way is the most efficient for them.

From an organizational standpoint, how do teams migrate to this new view of the world? LexisNexis has seen that with traditional organization structures, different teams report to different parts of the organization with different goals, which makes integrating those teams that much harder. Furthermore, other areas requiring change make tackling team and ownership aspects (such as manual tests and resource constraints) difficult. The developer's role is becoming less horizontal and more vertical, increasing responsibility but also empowering developers to understand their changes' impact.

6. Configuration Is Code

Continuous-deployment practitioners are finding that, at scale, not treating configuration like code leads to a significant number of production issues. Traditionally, configuration has been considered a runtime matter managed by an operations team or system administrators.

Changes are made to servers live, often in a one-off fashion that can lead to server drift. For example, an engineer is experimenting with optimizing query speeds for a database and changes the configuration on one of the database boxes. This change must be replicated to four database servers. When multiple servers are intended to represent the same application, having even one undergo configuration drift can lead to unknown or difficult-to-debug breakages.

Modern configuration management tools, such as Ansible, Puppet, Chef, and Salt, allow configuration management to be scripted and orchestrated across all server assets.

The new normal is that organizations should treat managing configuration the same as managing features and code. For example, at Netflix, for every commit, the build process creates a Debian package completely specifying the needed dependencies and then installs them in a new Amazon Web Services virtual-machine image.

The summit participants from Facebook and Netflix noted that despite tooling, configuration changes can still cause difficult-to-debug errors. Netflix does 60,000 of such changes daily and has no system for tracking or reviewing them. This leads to, as the Netflix participant put it, the company often shooting itself in the foot. Red Hat teams have found that, just as with large code bases, large configuration suites can become unruly.

Organizations should treat managing configuration the same as managing features and code.

The lesson from the companies at the bottom of the continuous-deployment spectrum is to consider configuration management right from the start of a new project or when transitioning projects with architectural baggage to a continuous-deployment model. In other words, configuration management should be a core competency that's treated like code. Treating configuration like code implies using all the best practices related to coupling, cohesion, continuous integration, and scale.

7. Comfort the Customer with Discomfort

As companies transition to continuous deployment, they're experimenting with ways to comfort customers regarding the new pace of delivery. In today's consumer world, as products and devices receive a constant stream of updates, customers often have no choice but to accept them. New generations of customers might, in fact, expect them. If mobile devices are training all of us to accept constant change, and if even cars and televisions are automatically updating themselves, why not business software? The number of customers willing to wait a year or two for updates will rapidly dwindle. Still, not all customers and companies are ready for this change.

One prominent example of this challenge involves Microsoft's experience with Windows 10. Microsoft has shifted from large, infrequent updates of its OS to regular incremental improvements. The effort to migrate users to Windows 10 has also been notably more proactive—for example, prefetching installation files, frequently prompting users to upgrade, and restricting their ability to opt out of updates. These changes might appeal to savvier customers but burden enterprise customers who might be unwilling or unable to accept frequent changes owing to those customers' internal integration testing and regulatory concerns.

IBM and Mozilla include important stakeholders in unit and integration tests during development. This reduces the risk of failed deployments on the stakeholders' premises and helps them feel more comfortable accepting new releases. Cisco has been exploring using more rapid deployments as a model for coinvention with customers.

Often, the biggest source of customer discomfort is the disrupted productivity when a customer upgrades versions. For example, IBM used to take a month to migrate a system to a new version at a customer's site. The primary challenge was coordinating code and database changes with on-premises instances. Eventually, IBM shortened the process to one hour. Similarly, at SAS, the biggest deployment barrier was that each deployment imposed long periods of downtime for customers and had to support many versions of datasets and deployed systems.

When moving speedily, companies must consider whether they're moving faster than users desire. Still, the best comfort a company can provide is the ability to deliver a change at a moment's notice, whenever the customer is ready.

When moving speedily, companies must consider whether they're moving faster than users desire.

8. Looking Back to Move Forward

Continuous deployment requires continuous reflection on the delivery process.

Almost every summit participant had a story about bringing down entire operations with accidental mistakes in configuration changes. For example, a malformed JSON (JavaScript Object Notation) setting once brought down the entire discovery component of Netflix's architecture.

To support reflection on production failures, all the companies employ *retrospectives* (or *postmortems*). In retrospectives, team members discuss the causes and consequences of an unexpected operation outage or deployment failure. They also discuss potential process changes.

Several participants described their experiences with retrospectives. At Netflix, developers report outages as issues in an issue tracker and rate their severity. By tracking outages, developers can perform a meta-analysis of them to uncover trends and systematic issues with deployment processes. More severe outages are discussed at weekly retrospectives, which are attended by multiple stakeholders across teams.

Some participants mentioned that despite retrospectives' usefulness, they can be dreadful. Developers find it hard to hear about a coding mistake's impact on users and have trouble factoring out emotions. Mentioning victories can help maintain team morale and ease raw emotions. In certain circumstances, it makes sense to leave the responsible party outside the room, if possible. Still, despite the potential unease during postmortems, several companies observed a considerable drop in errors after starting them.

Retrospectives can also shift cultural views. At Facebook's inception, the company instilled its developers with a culture of "Move fast, break things." However at a certain point, that message was taken too far, and a new moderating creed emerged: "Slow down and fix your s***."

Some of the other companies have reflected on the benefits of particular practices and have sought data verifying the benefits. For example, after extensively studying code reviews, Microsoft has so far found no significant defect reduction. Instead, the primary benefits involve knowledge sharing and improved onboarding (the process by which new employees learn the necessary skills and behaviors).

As companies continue to adopt continuous deployment, the need exists not only to calibrate how a practice is exercised and how a culture is defined but also to constantly question the benefits and effectiveness of having those practices and cultures in the first place. An important aspect of retrospectives is to maintain a "blameless culture," but this can be difficult when developers are also expected to be fully responsible for their deployed changes from cradle to grave.

9. Invite Privacy and Security In

Most summit participants indicated that privacy and software security were silos—the responsibility of a specific group and not the responsibility of all the developers involved in implementing deployable software. Continuous deployment might increase risk because privacy and security experts can't review every rapidly deployed change and because sprints often don't plan for security concerns.[8]

As we mentioned before, in waterfall and spiral development, developers throw the code over the wall to testers to deal with defects. With agile methodologies, testers are "invited to the table" and participate as partners from the beginning of an iteration. Together, then, the developers and testers throw their tested products over the wall to the operations team to deploy the product.

With continuous deployment, the operations team is also invited to the table, participating throughout an iteration and dealing the operations implications of feature development. However, the people in the security and privacy silos often aren't invited to the table. Since 2012, researchers have been discussing how to establish collaboration between security teams and development and IT teams.[9] This collaboration is usually called *DevSecOps*. We propose going further and explicitly inviting both privacy and security (PrivSec) folk to be involved throughout development (DevPrivSecOps). The aim is to increase the security knowledge of developers, testers, and operations staff and increase the partnership of privacy and security experts.

Companies can have a separate process or oversight for changes that have a higher security risk or privacy implications. At Facebook, a code change considered to have privacy implications might go through a push process that's longer than a daily one.[10] Additionally, a small team creates an access layer for all data and controls that forces adherence to privacy and regulatory concerns. Google has instituted controls for secure deployment, such as authorizing users who check in code for deployment, strict access control, and checksumming binaries. Google also has a strict division between its production network and company network. The production network consists of servers only; having no workstations reduces the possibility of tampering with deployed code.

10. Ready or Not, Here It Comes

Your competitor continuously adds value to its products. Do you? All the summit participants indicated the urgency of rapidly delivering new functionality to remain competitive.

A 2015 global survey by CA Technologies indicated that of 1,425 IT executives, 88 percent had adopted DevOps or planned to adopt it in the next five years.[11] DevOps and continuous deployment have similar practices; some people informally equate the two approaches. According to a 2015 Puppet Labs survey involving 4,976 respondents, IT organizations that adopted DevOps experienced 60 times fewer failures and deployed 30 times more frequently than organizations that hadn't adopted DevOps.[12] The respondents indicated widespread adoption of DevOps worldwide and in organizations of all sizes. The top five domains using DevOps were technology, Web software, banking and finance, education, and telecommunications. The prevalence and growth of DevOps is possible only if performance indicators support business benefits such as more customers, collaboration across departments, improved software quality and performance, and faster maintenance.[11,12]

Software engineering educators must also take notice. In the words of Brian Stevens, former executive vice president and chief technology officer at Red Hat, "The legacy model of software engineering just isn't going to survive this transition."[13] Software engineering education often lags behind the new reality of continuous deployment and focuses on the legacy model. Core undergraduate software engineering courses must also teach fundamental skills such as

- continuous integration and build,
- automated integration and system testing, and
- the need to follow good validation-and-verification practices if developers don't want to be awakened in the middle of the night to fix the code they deployed that day.

Additionally, educators must make undergraduates aware of the realities

ABOUT THE AUTHORS

CHRIS PARNIN is an assistant professor in North Carolina State University's Department of Computer Science. Contact him at cjparnin@ncsu.edu.

HARLEY BOUGHTON is a PhD student in computer science at York University. He previously was a software developer and the program manager for DB2 and dashDB at IBM. Contact him at boughton@yorku.ca.

ERIC HELMS is a developer at Red Hat Software and a PhD student in North Carolina State University's Department of Computer Science. Contact him at edhelms@ncsu.edu.

MARK GHATTAS is a senior solutions architect at Cisco Systems. Contact him at mghattas@cisco.com.

CHRIS ATLEE is a senior manager of release engineering at Mozilla. Contact him at chris@atlee.ca.

ANDY GLOVER is a manager of delivery engineering at Netflix. Contact him at aglover@netflix.com.

of deploying into a live, larger-scale environment, and the related concerns such as data migration, deployment strategies, deployment pipelines, and telemetry coding patterns.

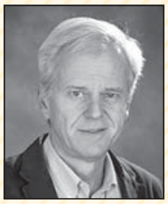
Ready or not, here comes continuous deployment. Will you be ready to deliver?

References

1. T. Savor et al., "Continuous Deployment at Facebook and OANDA," *Proc. 38th Int'l Conf. Software Eng. Companion*, 2016, pp. 21–30.
2. M. Marschall, "Transforming a Six Month Release Cycle to Continuous Flow," *Proc. 2007 Agile Conf.* (Agile 07), 2007, pp. 395–400.
3. R. Torkar, P. Minoves, and J. Garrigós, "Adopting Free/Libre/Open Source Software Practices, Techniques and Methods for Industrial Use," *J. Assoc. for Information Systems*, vol. 12, no. 1, 2011, article 1.
4. Z. Codabux and B. Williams, "Managing Technical Debt: An Industrial Case Study," *Proc. 4th Int'l Workshop Managing Technical Debt*, 2013, pp. 8–15.
5. M. Mäntylä et al., "On Rapid Releases and Software Testing: A Case Study and a Semi-systematic Literature Review," *Empirical Software Eng.*, vol. 20, no. 5, 2014, pp. 1384–1425.
6. J. Humble, *Lean Enterprise: How High Performance Organizations Innovate at Scale*, O'Reilly Media, 2015.
7. B.W. Boehm, Software Engineering Economics, Prentice-Hall, 1981.
8. Z. Azham, I. Ghani, and N. Ithnin, "Security Backlog in Scrum Security Practices," *Proc. 5th Malaysian Conf. Software Eng.* (MySEC 11), 2011, pp. 414–417.
9. J. Turnbull, "DevOps & Security," presentation at DevOpsDays Austin 2012, 2012; www.slideshare.net/jamtur01/security-loves-devops-devopsdays-austin-2012.
10. D.G. Feitelson, E. Frachtenberg, and K.L. Beck, "Development and Deployment at Facebook," *IEEE Internet Computing*, vol. 17, no. 4, 2013, pp. 8–17.
11. DevOps: The Worst-Kept Secret to Winning in the Application Economy, *CA Technologies*, Oct. 2014; www

ABOUT THE AUTHORS

JAMES HOLMAN is a senior director at SAS and the division head for deployment of SAS software. Contact him at james.holman@sas.com.



MICHAEL STUMM is a professor in the University of Toronto's Department of Electrical and Computer Engineering. Contact him at stumm@eecg.toronto.edu.

JOHN MICCO is an engineering-productivity manager at Google. Contact him at jmicco@google.com.

SHARI WHITAKER is the manager of development operations at LexisNexis. Contact her at shari.whitaker@lexisnexis.com.

BRENDAN MURPHY is a principal researcher at Microsoft. Contact him at bmurphy@microsoft.com.

LAURIE WILLIAMS is a professor and the associate department head in North Carolina State University's Department of Computer Science. Contact her at lawilli3@ncsu.edu.

TONY SAVOR is an engineering director at Facebook and an adjunct professor in the University of Toronto's Departments of Computer Science and Electrical and Computer Engineering. Contact him at tsavor@fb.com.

.ca.com/us/~/media/Files/whitepapers/devops-the-worst-kept-secret-to-winning-in-the-application-economy.pdf.

12. *2015 State of DevOps Report*, white paper, Puppet Labs, 2015; https://puppetlabs.com/2015-devops-report.
13. B. Stevens, "2014 Red Hat Summit: Brian Stevens, Red Hat Keynote," 2014; www.youtube.com/watch?v=8B56mdobgZE.