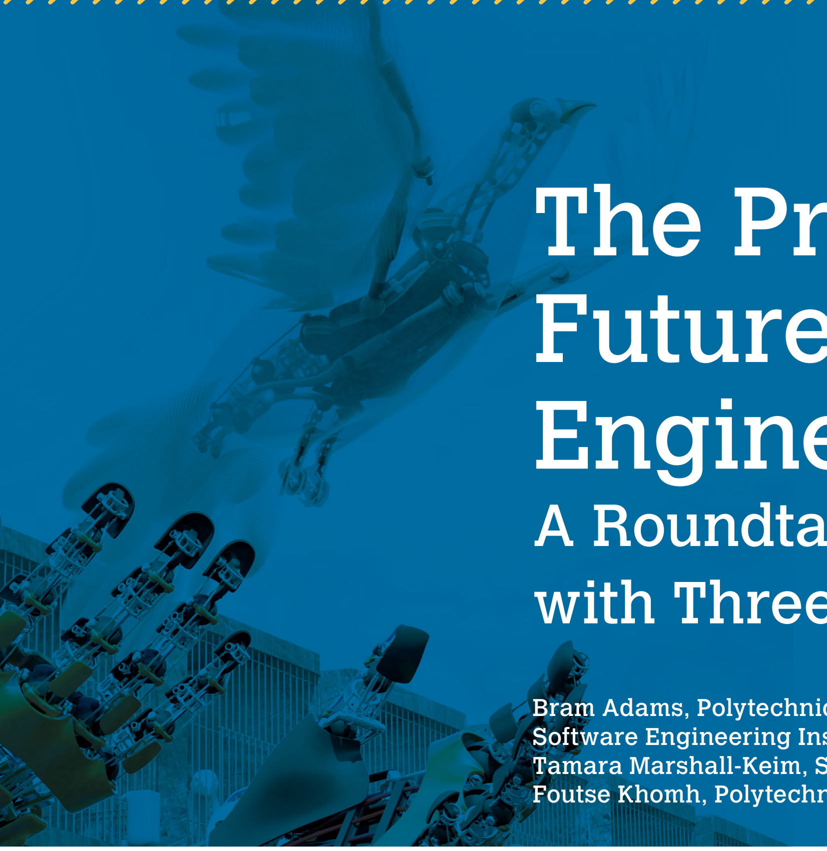

The Practice and Future of Release Engineering

A Roundtable with Three Release Engineers

Bram Adams, Polytechnique Montréal // Stephany Bellomo, Software Engineering Institute // Christian Bird, Microsoft Research // Tamara Marshall-Keim, Software Engineering Institute // Foutse Khomh, Polytechnique Montréal // Kim Moir, Mozilla

RELEASE ENGINEERING focuses on building a pipeline that transforms source code into an integrated, compiled, packaged, tested, and signed product that's ready for release. The pipeline's input is the source code developers write to create a product or modify an existing one. Enterprises running large-scale websites and delivering mobile applications with millions of users must rely on a robust release pipeline to ensure they can deliver and update their products to new and existing customers, at the required release cadence.

This special issue provides an overview of research and practitioner experience, and this article in particular aims to give you insight into the state of the practice and the challenges release engineers face. It features highlights from interviews with Boris Debic, a privacy engineer (and former release engineer); Chuck Rossi, a release-engineering manager; and Kim Moir, a release engineer. We asked each of them the same questions covering topics such as release-engineering metrics, continuous delivery's benefits and limitations, the required job skills, the required changes in education, and recommendations for future research.

Every product release must meet an expected level of quality, and release processes undergo continual fine-tuning. What metrics do you use to monitor a release's quality? Do you roll back broken releases after deployment? If so, how?

Debic: Our main measures are threefold: the number of open bugs ranked by priority, the number and percentage of successful releases, and the number and percentage of releases that are abandoned late in the game. The first two measures allow us to gauge the overall release health of a service; the third measure can uncover issues in the testing pipeline or growing code complexity. We track these metrics and make comparisons from quarter to quarter.

Related to testing, another metric is the greenness of the testing pipeline. Many tests, from code to performance tests, are run daily in a continuous fashion. Stability of tests is a signal of product maturity and good engineering practices. Despite some arguments to the contrary, this measure effectively increases the velocity of product development and release.

We also track a host of more fine-grained metrics. Every step—with its duration, outcome, operation, logs, arguments, and other relevant details in execution and setup—is logged for every release that runs at our company. Refinements in the release system are direct results of observing patterns and quantifying

0740-7459/15/$31.00 © 2015 IEEE

different processes, tools, and approaches using this dataset.

To gain another perspective, we have systems that interact with our users, either by providing them a way to give direct feedback or by going through logs and looking for different types of failures. This data is distilled and presented to product teams as a collection of signals that speak of product robustness and of complaints that users mention most often.

For Web services and servers, "canarying" is another key component of successful releases. Canary rollout strategies depend on the type of service, user expectations, and contractual obligations. In this type of rollout, we gradually increase the exposure of the new binary and at all times monitor the critical parameters. Canaries are the bread and butter of the final stages of a well-designed release process.

Rossi: I'll talk about Web deployment first and then contrast it with mobile deployment. For Web deployment, we use the metrics of the code going into the master branch, the test results, and performance lab results. The next level includes metrics for products being released, such as core tests, unit tests, and performance experiments like time to interaction (TTI), fatal-error rates, the number of errors per page, and any new errors that we hadn't seen in the production logs.

Then comes the canary step. The set of binaries for a release sit in the canary state for 30 minutes to an hour. I look at the logging and flag new errors, error rate changes, and fatal or elevated error rates for an existing error. Core metrics include TTI; the number of likes, photos uploaded, and comments; and the amount of tagging. We compare the growth and interaction metrics from the canary to those from production. A release engineer and the developers look at them with more detailed dashboards. For example, the ad teams have dashboards on ad displays and ad click-throughs.

Our alerting system works on either absolute numbers or the percentage rate. The biggest alert for the Web is the log data for each new build. In a canary, we collect that log data separately from the regular production traffic. A website has thousands of firing errors and warnings, and we look for changes in those. An analysis of errors in the log data that differ from production is the first part of the canary. That's easy, and it's universal—it doesn't matter what your app is doing.

Another big alert is when the canary TTI is much higher than the production TTI. Is it because we just increased login calls by 10 times? Or because a database call to render the first page is not going through cache and it's trying to do a lookup every time? TTI helps us flesh out the problem. We pay particular attention to how long it takes the main page to render.

I have graphs for the back-end machines, but I look for effects on the front end. When I see those effects, I'll start digging down. I might see, for example, that only Internet Explorer 7 on Windows boxes is showing a bad TTI. Or I'll realize that the front end is rendering so slowly because I've lost half the back-end machines that are providing data for this service. I wouldn't have found that internally, but I will find it in canary because it is millions of people.

Mobile deployments are more challenging than Web deployments because we don't own the ecosystem, so we can't do all the things that we would normally do. And the canaries are huge. We watch cold start, warm start, the app size, and the numbers of photos uploaded, comments, and ads being displayed or clicked. Growth and engagement numbers and the crash rate are important to the company. If the crash rate fluctuates, we immediately take action to understand why.

Concerning rollback, we've never had a canary that bad. Generally, it's always rolling forward. We'll promote the release candidate to the production binary in our store, roll it to 5 percent of users, and get data back from that. If that 5 percent looks good, we'll roll it out to the rest of the population. I always make the analogy that it's like a bullet from a gun. It just keeps going. The mobile ecosystem is so broken when it comes to software management that I don't want to force people to re-download. Every time I have to ask them to re-download, I lose a certain percentage of people who just never do it. So that's the challenge.

For Web services and servers, "canarying" is a key component of successful releases.

Moir: At Mozilla, release engineers don't monitor the quality of the release; we have a team called Release Management to perform that function. We use a "train model" for managing releases. When developers have a new feature, they'll land it on a certain branch and make sure the test suites run green. If so, the change set will be uplifted to another branch to ensure that the patch integrates with other changes on that branch and tests run green. Eventually, the new feature reaches the Aurora branch, which is an alpha branch, where it will sit for six weeks to bake; then it goes to the beta branch. Finally, six weeks later it goes to the release branch. This is one way of ensuring stability.

We limit the number of people who get a release. On a given release day, we might let 5 percent of the population running the desktop version of our browser get the new release. We have automatic crash reporting in the browser that reports to databases here. How many crashes occurred? Are certain operating systems, platforms, or add-ons having problems? We'll analyze answers to those questions to determine whether we can roll the release out to the rest of the population. Other metrics come from users who give us feedback during the beta, support requests on our support website, sentiment analysis on Twitter, and the top 10 crashes across our continuous integration every week.

Concerning rollback, we don't really roll releases back. If there were a serious problem, like a huge number of crashes on a certain release, we would block it so that no updates would occur and then do a point release. For example, if there were a security issue causing problems, we would do a point release so that users wouldn't get the last release and would be automatically updated to the newer release with the security fix. We call this a "zero-day fix."

Amid all the hype and buzz about continuous delivery, what's currently possible, and what are the limitations? How far should you go with continuous delivery?

Rossi: I've never worked in a true continuous-deployment environment. We have a pseudo-continuous deployment here—it's twice a day. Size is the limiting factor. All the continuous-deployment places I've visited had engineering teams of 20 to 50, even 100 people, pushing to a website with a number of users at best in the double-digit millions per day. The same processes don't scale above a few hundred developers working on a common code base or to a website that has either more complexity or users into the hundreds of millions. It doesn't scale at our company's size. Continuous deployment works for small teams, with 20 to 30 changes per day.

Mobile deployments are more challenging than Web deployments because we don't own the ecosystem.

Continuous deployment obviously shines in the Web area, where you own the ecosystem. You can publish effortlessly to your Web fleet, and your users get the fixes and features instantly without noticing it. In minutes or hours, you will know whether something is wrong with the release.

As I understand continuous delivery, it will not happen on mobile in the near future. The current app distribution system is based on an ancient model that's not even as good as shrink-wrapped software. In this model, you build an artifact, you put it out to a third party that has total control over when and how it gets out, then the end users constitute a completely disparate map of if, when, and how it gets updated. And there is no way for you to influence that ecosystem.

So, you need user interaction for every single update, and that's insane. Why should I have to take time out of every day for the rest of my life to push a button and have my phone update its apps? But that's the model that we've had with iOS. iOS 7 has an easy way to turn on automatic app updates. Then you're not seeing that double-digit red number on your App Store icon every day. Unfortunately, though, this feature is not on by default. Android will put up roadblocks even if you have auto-update turned on. Of course, the owners of the platforms have valid reasons for trying to maintain this control, such as preventing malicious apps from auto-installing.

Our company, which has both good infrastructure and complex apps, can do automatic updates. In fact, any mobile developer could provide users the infrastructure to use their own channels to update apps.

Software deployment professionals need a solution that addresses the security concerns but lets us update our valid, legitimate apps seamlessly with no pain to the user.

Moir: I think the continuous-delivery model for desktop software works well if the updates are silent and users don't get constantly notified about them. Otherwise, they get annoyed. As Chuck said, the mobile model obviously is different because Google and Apple own the distribution, and the default behaviors require users to update as they feel like it.

At our company, we are focused on relentlessly automating everything. We're automating the uplift of all the changes from beta to release or from Aurora to beta, to have fewer manual steps. We've come a long way from when we first started releasing software, and we had a big page of instructions to follow by hand, which was not very efficient. Now there's a great deal of deep knowledge about how everything works, so that when something goes wrong, we can fix it. This lets us focus on writing tools to improve our continuous-integration farm and our release automation.

If a company is thinking about moving to continuous integration, it needs to get a release engineer on board in the early stage, not the later stage. Sometimes, product teams work on a product almost in secret and throw it over the fence when they're done. Then, release engineers want to run away screaming when they see that the product is built on a hacky pile of spaghetti, and they have to fix it.

It's also good to have someone who's not emotionally attached to the code and who is focused on getting the pipeline in place as well as getting the product in place. The release engineer doesn't get upset if you say, "You can't put that feature in because it's going to break everything, and we need to ship tomorrow." As a release engineer, your focus is getting a stable release out the door.

Debic: The possibilities and limitations of continuous delivery depend on the type of deliverable. Is it a Web service, a mobile application, or software for a medical device or aircraft autopilot? In the high-tech business, the Holy Grail of release engineering is something called "push-on-green." As soon as a developer has committed a change list to the code base, it automatically gets into a pipeline that tests, executes, and canaries the change list. If all of the elements pass the change list, it goes into production. Push-on-green does not always make sense: a change list may be dependent on a set of change lists. There may be dependencies between functional parts of the product or between services. It may be impossible to immediately deploy the change—think of mobile devices that have a wholly different model than a service in a datacenter. Users may not want to interrupt their days, or the change may not be compatible with all devices.

Often, people unfamiliar with release engineering don't understand the inherent complexity of transforming code into a form that's tested, deployed, signed, and reproducible. How do you educate others about the value that release engineering brings to a team?

Moir: In my current environment, release engineering is definitely well received. Because if you can't build, we can't ship. And if we can't ship, we don't get paid. In other companies, I've seen that release engineers are second-class citizens, or they are expected to perform miracles with no advance warning or resources.

In those cases, obviously some education is necessary. And maybe it stems from the fact that release engineering is not taught in school as a discipline, so people aren't exposed to it. I like to help spread the word by writing about release engineering on my blog. And I've helped organize workshops for release engineering to try to bring the community together.

In his book about remote work, *A Year without Pants*, Scott Berkun writes about his time as a manager at Automattic, which developed WordPress.com. At Automattic, all new hires spend a few weeks in a support role, which gives them a better understanding of customer issues and the overall process to get software out the door.

DevOps (development and operations) is another practice that breaks down the walls between operations and development. If you

A company needs to get a release engineer on board in the early stage, not the later stage.

give developers the responsibility not only to land code but also to make sure that it actually works in production and that the customer is happy, they become more aware of the whole pipeline of moving software from development to the customer. And if something does not work, developers are involved with backing it out and writing patches to fix it.

Debic: If I need to describe release engineering to colleagues who do not know my work from firsthand interactions, I tell them that release engineering is the difference between manufacturing software in small teams or startups and manufacturing software in an industrial way that is repeatable, gives predictable results, and scales well. These industrial-style practices not only contribute to the growth of a company but also are key factors in enabling growth.

A release engineer has a special mind-set. We look at everything that is going on in a tech company, and we try to industrialize the process. Where others see features, we see release challenges. Where others count change lists, we count how long it took for a change from the time it was submitted until the time it was in front of the customer. While others add people to a project, we look at how the added complexity will affect it.

If you can't build, we can't ship.
And if we can't ship, we don't get paid.

Rossi: I've always maintained that if you're a good release engineer, you can work for any software company in the world. My wife makes fun of me because I come home and tell her, "Oh, you know, Sylvia, three recruiters contacted me today. I'm thinking I'm hot stuff." And she says, "Yes, I told you no one wants to do what you do."

There's a conception that release engineering is nasty work. When I started doing this, my first job was with IBM in 1988. I worked on the release of an integration of two operating systems. This is really complex—a huge software project with two massive things that intersect, and it has to be reproducible, repeatable, and testable. No one gets this exposure until they're dropped in the middle of it and have to react to it. And you'll find many good release engineers who are release engineers now because they started in a company where no one would do it. That's traditionally how people have fallen into this role.

I don't think you have to make the value proposition to any company of why you want someone doing release engineering, especially since there has been movement in the continuous-integration and continuous-delivery worlds in the past. I've talked to small startups, and generally it's not one of the first things they're worried about. But once they start to grow, they begin to look for a person to do release work.

Universities and colleges don't explicitly teach release engineering as a discipline. Given this limitation, how do you find good release engineers to hire? Should curricula change to include these skills? If so, what courses would be essential?

Debic: This is a very good question. Release engineering is not taught; it's often not even mentioned in courses where it should be mentioned. I think the main reason is that the release-engineering practice itself has been hard to define. As you see from the answers of your other guests, the approaches are quite diverse in nature and scope.

But perhaps we should not have skills-based curricula for release engineering anyway. At Google, release engineers are software engineers; there is no difference. The complexity of work they do and the tools they use are the same as for product engineers. Certain establishments treat release engineers and quality assurance engineers differently, but this is a short-sighted strategy, and my company is proof that the opposite works better.

Rossi: I've spent some time trying to work through this both at the university level to get the curriculum lined up and at a personal level. One of my biggest hiring concerns is that I need to hire release engineers. It's like finding unicorns. I look for a strong technical background and experience with programming on either the product side or the infrastructure side. I want utilitarian programmers and people who get stuff done in the realm of system administration or tool writers. I don't need top-notch C++ programmers or people concerned with the delicacies of optimizing C algorithms.

The next thing I want is architecture knowledge. I want people to understand large-scale, multitiered, distributed systems well enough to debug or get into a system and see where it is falling over. If you're release engineering for the full stack, you're pushing everything from the databases, the back-end systems, the caching layer, the front end, the Web servers, and everything in between. You need to know how it all works if you're the one rolling it out.

Then I look for release engineering proper. Release engineers understand where there's risk in making things reproducible, repeatable, and able to go back to any state of what was built. Experience tells them when you don't make this kind of change at this point in the cycle because it's too great a risk—that's hard to teach. Release engineers are familiar with the source control systems, and they can do surgery on trees and branches; flatten conflicts; and safely deploy a new binary, drain existing connections, and bring up new services seamlessly.

If people come to me from a job where they had maintenance windows for rollouts, that's a joke. I'm not going to take you seriously if you're from a context where you can't use your bank between midnight and 3 a.m. because it is down for maintenance. That's just not acceptable from a release-engineering standpoint.

Moir: Release engineers are hard to find, and one problem is that they don't teach the skill set in school. I recently looked at the undergrad classes required to graduate with a computer science degree from a major university, and I was struck by how much of it was theory and not much was practice and deploying code. In most computer science programs, there is little emphasis on infrastructure. I think the expectation is that students will learn the practical aspects later.

It would be great if schools taught release-engineering skills, but what classes would you remove from a computer science curriculum to accommodate this? Still, some topics that I would like to see in a course are version control systems, like cloning, branching, and merging; bug-tracking systems; writing patches and testing them against existing code bases; and interacting with people. Other skills would be how to maintain continuous-integration deployment and infrastructure and how to set up a release pipeline. Case studies of how large companies do release engineering would be useful. *Continuous Delivery*, by Jez Humble and David Harley, could be an excellent textbook for such a class.

Release engineers are hard to find, and one problem is that they don't teach the skill set in school.

What should researchers focus on regarding release engineering, continuous delivery, and related topics? Where can research contribute to problems you see with release engineering or continuous delivery?

Moir: One thing I struggle with is how to model the capacity I need for our continuous-integration farm. For instance, yesterday we ran 3,200 build jobs and 74,000 test jobs, and each test job ran performance tests or correctness tests. It's an active and complicated environment, and I would love to know—given *x* number of platforms, *y* number of branches, and the matrix of tests and builds we run on each of these platforms—if we increase our number of commits, how much additional capacity will we need within the next year?

We could also model large continuous-integration farms for the possible effects of reducing the number of tests run. We could use an algorithmic model that you can plug in and enter parameters such as the type of machines running, the environment, the number of builds, and other constraints. And the model would show where your limits on capacity might be.

Another issue is high pending counts. We have a lot of jobs waiting for machines, and it seems like we're always playing Whac-A-Mole on the bottleneck. We run almost all tests on all commits, but do we really need to go to that effort and expense? How can we break out only the relevant tests that need to be run on a given commit so that we use our capacity more efficiently? Some dependence analysis tools would be useful, to trace through the code, map code changes, test coverage, and thus invoke only the relevant tests for that code.

Rossi: This doesn't really apply in a pure continuous-delivery situation, but in a near-continuous-deployment system, I would like to know the velocity of code change over time. Does the change rate narrow down to a point, or does it ramp up as the date gets closer? Does the defect rate increase or decrease as we get to that end point? Because, if I do analysis on my cycles, and I see that two or three days before the final release I'm getting a twofold increase in the number of changes, this indicates risk. And risk has increased at the worst possible time, at the end of the cycle. As a release engineer, you always feel like you're cramming in stuff at the last minute, when you're trying to have time to settle, let the metrics come in, and get what we call our "soak period." But we often can't do it because we're taking changes right up to the moment that we release. Is it really always this mad dash at the end?

The development cycle would make an excellent subject for analysis too. What is the effect on code delivery and the defect rate of two-, four-, or six-week cycles? This is very relevant to mobile. Does a quicker release cycle in mobile produce better and less buggy products?

Debic: An escalating number of computer software applications, systems, and home-electronics products are permeating all industries. This results in an exponential growth of programmable entities. On the other side, we have the output of computer science schools, which is growing linearly and slowly. The gap between the work to support this growth and the number of qualified professionals is growing. What can we do about this gap? People are trying different approaches. My colleague Peter Norvig is working on expanding the workforce through online education. MOOCs (massive open online courses) are available, and people are taking advantage of this new, unprecedented channel. Ray Kurzweil is more pragmatic. He is building a computer—an AI, really—that programs itself. And my colleague Sinisa Srbljic thinks that the best way forward is to build a platform that consumers can use to customize applications by themselves, without formal knowledge of computer science.

How can we break out only the relevant tests that need to be run on a given commit?

If a system is well engineered, it should be adaptable to its environment and perhaps even learn from it. Right now, too much software change happens as a result of humans banging on keyboards, and then we have to release all of that. We are running out of programmers, so software in the long term will have to be either more adaptable by design or written in such a way that consumers can change and adapt it. This would change our model of computing to include consumers as also modifiers, creators, contributors, and editors of software.

Release engineering is a complex field with many approaches to ensuring that quality software can be released on a predictable schedule. Company culture regarding release engineering's importance, infrastructure and tooling investment, and commitment to continuous delivery varies widely among enterprises. Similarly, the scope of a release engineer's role depends on where she or he works, the number of products to build, the operating systems and platforms on which they're deployed, and the release cadence. This roundtable raises many interesting areas for research and for improving education to ensure that future software developers better appreciate the scope and challenge of release engineering.

The seven articles in this special issue benefit developers in two ways. The first group of articles reports on the experience of companies who migrated toward rapid or even continuous release schedules. In "Continuous Delivery: Huge Benefits, but Challenges Too," Lianping Chen discusses the benefits and challenges of continuous delivery at Paddy Power. Martin Michlmayr and his colleagues investigate release planning's importance for open source systems in "Why and How Should Open Source Projects Adopt Time-Based Releases?" In "The Highways and Country Roads to Continuous Deployment," Marko Leppänen and his colleagues examine Finnish industry's adoption of continuous deployment. "Achieving Reliable High-Frequency Releases in Cloud Environments," by Liming Zhu and his colleagues, discusses reliability issues related to high-frequency releases in the cloud.

The second group of articles focuses on release engineering's specific challenges. "Release Stabilization on Linux and Chrome," by Md Tajmilur Rahman and Peter Rigby, reports on an empirical study of the time and effort involved in release stabilization on Linux and Chrome, whereas "Rapid Releases and Patch Backouts: A Software Analytics Approach," by Rodrigo Souza and his colleagues, examines how the release process changed when Mozilla transitioned to rapid releases. Finally, Jonathan Bell and his colleagues propose approaches to speed up testing of Java projects in "Vroom: Faster Build Processes for Java." We hope these articles convey an idea about the state of the practice and the challenges of release engineering today.

ABOUT THE AUTHORS

BRAM ADAMS is an assistant professor at Polytechnique Montréal, where he heads the Lab on Maintenance, Construction, and Intelligence of Software. His research interests include software release engineering in general, as well as software integration and software build systems in particular. Adams received a PhD in computer science engineering from Ghent University. Contact him at bram.adams@polymtl.ca.

STEPHANY BELLOMO is a senior member of the technical staff at the Carnegie Mellon University Software Engineering Institute. Her research interests include incremental software development, the architectural implications of DevOps, and continuous integration and delivery. Bellomo received an MS in software engineering from George Mason University. Contact her at sbellomo@sei.cmu.edu.

CHRISTIAN BIRD is a researcher at Microsoft Research in Redmond, Washington. His main research interest is empirical software engineering, predominantly examining collaboration and coordination in large software teams in both industrial and open source contexts. Bird received a PhD in computer science from the University of California at Davis under advisor Prem Devanbu. Contact him at christian.bird@microsoft.com.

TAMARA MARSHALL-KEIM is a senior editor at the Carnegie Mellon University Software Engineering Institute. Her research interests are applied linguistics, theory of rhetoric, and computer studies in language and literature. Marshall-Keim received an MA in English from the University of Florida. Contact her at tmarshall@sei.cmu.edu.

FOUTSE KHOMH is an assistant professor at Polytechnique Montréal, where he leads the SWAT (Software Analytics and Technologies Lab) team on software analytics and cloud-engineering research. His research interests include software maintenance and evolution, cloud engineering, service-centric software engineering, empirical software engineering, and software analytics. Khomh received a PhD in computer science from the University of Montreal. Contact him at foutse.khomh@polymtl.ca.

KIM MOIR is a release engineer at a Mozilla. Her research interests are build optimization, scaling large infrastructure, and optimizing build and release pipelines. Moir received a Bachelor of Business Administration from Acadia University. Contact her at kmoir@mozilla.com.