MARCH/APRIL 2015

WWW.COMPUTER.ORG/SOFTWARE

IEEE Software

CODE INFLATION // 10

MANAGING TECHNICAL DEBT // 22

IEEE computer society

Special Issue on Refactoring: Accelerating Software Change

Submission deadline: 1 Apr. 2015 • Publication: Nov./Dec. 2015

Modern software is rarely written from scratch. It usually incorporates code from previous systems and is itself reincarnated in other programs. Software also isn't static; it constantly changes as bugs are fixed and features added. Usually these changes are performed by more than one programmer, and not necessarily by the code's original authors.

Refactoring supports this highly dynamic software life cycle. Basically, refactoring improves a piece of code's internal structure without altering its external behavior. You can use it to clean up legacy code, to understand a program, and as a preparation for fixing bugs or adding features. Although any behavior-preserving change can be considered a refactoring, many particularly useful and frequently recurring refactoring operations have been identified and catalogued. Over the past decade, popular development environments have started providing automated support for common refactorings, making refactoring less tedious and error-prone.

So, we solicit submissions for this special issue that focus on the real-world application of research, practical experiences, success stories, and lessons learnt in refactoring. Submissions should focus on one or more of these categories:

- experience applying refactoring tools to industrial code, including rigorous analysis of opportunities and challenges when using them;
- industrial experience (for example, good practices and lessons learned) in implementing or managing refactoring in specific application domains (for example, aerospace, banking, mobile, and embedded systems) or domains not traditionally discussed in the refactoring literature (JavaScript, mobile applications, architecture, and so on);
- research papers describing state-of-the-art processes and tools that enable, support, or improve refactoring, with evidence of their use and impact in industrial settings;
- empirical studies of refactoring "in the field," addressing one or more human, technical, social, or economic issues through qualitative or quantitative studies; and

studies of refactoring economics, including estimation and measurement of the size, cost, benefits, time frame, and quality for planning and controlling refactoring in actual organizations.into practice.

Questions?

For more information about the focus, contact the guest editors:

- Emerson Murphy-Hill, emerson@csc.ncsu.edu
- Peter Sommerlad, psommerl@hsr.ch
- Bill Opdyke, opdyke@acm.org
- Don Roberts, don.roberts@gmail.com

Submission guidelines

Articles should have a practical orientation and be written in a style accessible to practitioners. Overly complex, purely research-oriented or theoretical treatments aren't appropriate. Articles should be novel. IEEE Software doesn't republish material published previously in other venues, including other periodicals and formal conference or workshop proceedings, whether previous publication was in print or electronic form.

Manuscripts must not exceed 5,400 words including figures and tables, which count as 250 words each. Submissions exceeding this limit might be rejected without refereeing. The articles we deem within the theme's scope will be peer-reviewed and are subject to editing for magazine style, clarity, organization, and space. We reserve the right to edit the title of all submissions. Be sure to include the name of the theme or special issue.

Full author guidelines:
www.computer.org/software/author.htm
Submission details: software@computer.org
Submit an article:
https://mc.manuscriptcentral.com/sw-cs

IEEE Software
TABLE OF CONTENTS
March/April 2015 Vol. 32 No. 2

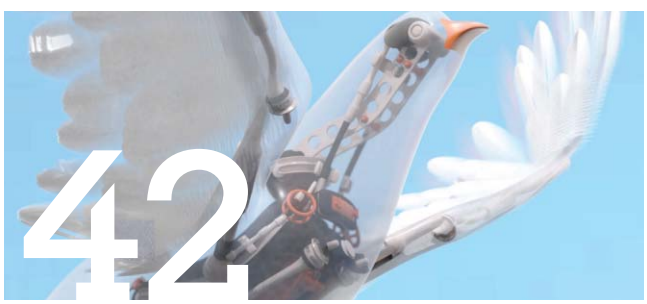
MULTIMEDIA

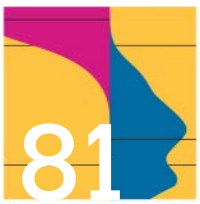
FOCUS

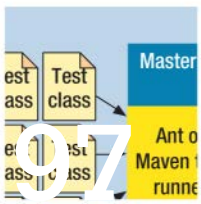
RELEASE ENGINEERING

FEATURES

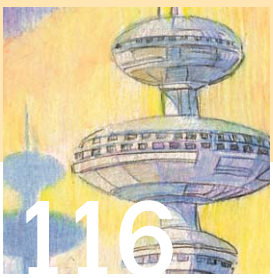
INSIGHTS







See www.computer.org/software-multimedia for multimedia content related to the features in this issue.



DEPARTMENTS

MISCELLANEOUS

FROM THE EDITOR

Editor in Chief: **Diomidis Spinellis**
Athens University of Economics and Business, dds@computer.org

The Strategic Importance of Release Engineering

Diomidis Spinellis

RELEASE ENGINEERS are building pipelines to deliver software products from the developers to the end users. This is the apt metaphor that John O'Duinn used to define release engineering. O'Duinn, who was Mozilla's director of release engineering for more than six years, was delivering a keynote on release engineering's value as a force multiplier at the 2014 USENIX Release Engineering Summit. He nailed the point.

Some used to consider release engineering a mundane activity, one that appealed to control freaks wishing to play God with other developers. How times change! In our era, where all nontrivial devices are controlled by software, more than a billion consumers can buy apps with a touch of a button, and software is delivered or served instantaneously through the Internet, release engineering has become strategically important. It affects the software we build, how we build it, and how we can make money out of it.

> A timely introduction of a clunker and a delayed entry of a masterpiece can destroy a product's chances of success.

Making Software ...

First consider agility. Reaping its many benefits requires smoothly running release-engineering machinery. Significant practices of agile software development include the coevolution of requirements and solutions, early delivery, and rapid, flexible response to change. These practices depend on the effortless creation of frequent software releases. Release plans that follow the "release early, release often" principle can increase a team's velocity, resulting in a beneficially tight feedback loop between developers and users. In some cases, concurrent, real-time, alpha-beta testing of new product features across slightly different releases can become a powerful oracle to guide product evolution.

Then think about software quality. The adroit introduction of features in new releases can improve the software's functionality, usability, and efficiency. In contrast, a bungled release can spell disaster. Dexterous software stabilization and issue triaging as part of release engineering will balance the new features with the required level of software reliability, maintainability, and portability. (Portability is again becoming important in cross-platform development markets, such as those for games and apps.)

Finally, take into account developer productivity. Inept release engineering can frustrate developers with protracted code freezes, agonizingly slow and brittle builds, and tiny infrequent time windows in which to commit their code to a production branch. This state of affairs is like pouring tar on developers' keyboards. Worse, besides holding back progress, it saps staff morale and increases turnover. Many sickly software shops have open release-engineering wounds.

... And Making Money

For highly innovative products and those facing cutthroat competition, time to market can make or break a launch. With potential customers globally connected through social networks like never before, firms rarely get a second chance to correct botched moves. Both a timely introduction of a clunker and a delayed entry of a masterpiece can destroy a product's chances of success. Adept release engineering can balance time pressures

0740-7459/15/$31.00 © 2015 IEEE

WELCOME NEW BOARD MEMBER AND DEPARTMENT EDITORS

I'm honored to welcome Marian Petre to the Editorial Board as our new associate editor in chief for human factors and also to congratulate Rafael Prikladnicki on his transfer from the Advisory Board to the Editorial Board as the department editor for the Voice of Evidence column. In addition, I'm excited to have Cesare Pautasso and Olaf Zimmermann serving as coeditors of the Insights column.

Marian Petre is a professor of computing in the Faculty of Mathematics and Computing at the Open University. She is also a past director of its Centre for Research in Computing. She has made contributions not only to software engineering research but also to software education and skill development for young researchers. She has also served as a guest editor for an *IEEE Software* special issue.

Rafael Prikladnicki is an associate professor at the Pontifica Universidade Catolica do Rio Grande do Sul and the director of TECNOPUC, the university's science and technology park. In this position he leads work at the intersection of academia and commercial software development. He is also leader and cofounder of the MuNDDoS Research Group on Distributed Software Development.

Cesare Pautasso is an associate professor at the Faculty of Informatics at the University of Lugano. His research group focuses on building experimental systems to explore the architecture, design, and engineering of next-generation Web information systems. His teaching, training, and consulting activities cover advanced topics related to emerging Web technologies, RESTful business process management, and cloud computing. Pautasso is a senior IEEE member and an advisory board member of EnterpriseWeb.

Olaf Zimmermann is a professor and institute partner at the Institute for Software at the University of Applied Sciences (HSR FHO) in Rapperswil. His areas of interest include Web-based application and integration architectures, SOA and cloud design, and architectural knowledge management. He is an author of *Perspectives on Web Services* (Springer, 2003) and contributed to several IBM Redbooks, including the first one on Eclipse and Web services (2001).

Please join me in congratulating Marian, Rafael, Cesare, and Olaf in their new positions.

with demanding software quality requirements, ensuring that the right product is launched at the right time.

Then comes marketing. Its executives who cover product packaging with sparkly "New!," "Improved!," and "Better Tasting!" labels are surely onto something. In demanding marketplaces, products that don't move forward sufficiently fast will stall and crash. Release engineering can assist marketing though the regular, reliable delivery of new features from a product's evolving roadmap. If you think this task is easy, consider that at the same time, current and legacy versions must also be serviced through bug fixes and urgent security patches. When done right, release engineering promotes openness by replacing vacuous "The Greatest Ever!" claims with concrete, accurate lists of features and fixes (and gotchas) associated with each new product version.

Crucially, release engineering can also form the basis of an entire business model. Some companies derive most of their revenue not from initial purchases but from product updates or subscriptions. In these cases, release engineering isn't supporting the product, release engineering *is* the

product. An important niche in this area is updates for software assets other than code, which must also be put under release engineering's control: virus definitions, football game player data packs, playlists, fonts, clipart, and so on. Business models based on frequent updates keep customers engaged and loyal, reduce the risk from shifting markets, and, by rapidly delivering features, allow for customer-driven innovation.

Finally, release engineering affects client relations and the company and product image. A slipshod release can lead to data loss and service disruption among customers or, in extreme cases, transform working appliances into useless bricks. On the other hand, a lack of visible progress in a software product can taint it as "abandonware." Few companies can successfully deal with the fallout and escape the damage caused by the consequent reputation damage. Timely, reliable, new releases keep happy those customers who live on the leading edge, while painless fixes and backward compatibility maintain a warm fuzzy feeling with the more conservative group.

Challenges

If you think the demands on release engineering are tough, wait till you see some additional challenges. In some application domains, regulators can prescribe software requirements, time schedules, and a lengthy, cumbersome approval process for new software releases. So-called 0-day security threats exploit vulnerable applications the same day the vulnerability becomes known, pressuring release engineers for a lightning-fast reaction. Then, there are users who, rightly, hate the disruption of frequent updates and want to be kept forever on long-term support branches (properly maintained, of course).

Add to this mix a rapidly evolving ecosystem of hardware, standards, operating systems, APIs, libraries, databases, and middleware, with its own unique pressing demands for maintaining compatibility. Supporting simpler embedded devices isn't easier because these might offer limited or intermittent connectivity and are often tended by untrained users—a ticking-time-bomb combination. Finally, within their organization, release engineers must support legacy tools and even hardware that might be required to build and maintain older versions.

Coming out with methods to address these challenges will probably require a lot more work and another *IEEE Software* theme issue in a few years. However, one thing is certain: release engineering is anything but boring; indeed, it can be an organization's hidden strategic asset.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.

HOW TO REACH US

WRITERS

For detailed information on submitting articles, write for our editorial guidelines (software@computer.org) or access www.computer.org/software/author.htm.

LETTERS TO THE EDITOR

Send letters to

Editor, *IEEE Software*
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
software@computer.org

Please provide an email address or daytime phone number with your letter.

ON THE WEB

www.computer.org/software

SUBSCRIBE

www.computer.org/software/subscribe

SUBSCRIPTION CHANGE OF ADDRESS

Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software*.

MEMBERSHIP CHANGE OF ADDRESS

Send change-of-address requests for IEEE and Computer Society membership to member.services@ieee.org.

MISSING OR DAMAGED COPIES

If you are missing an issue or you received a damaged copy, contact help@computer.org.

REPRINTS OF ARTICLES

For price information or to order reprints, email software@computer.org or fax +1 714 821 4010.

REPRINT PERMISSION

To obtain permission to reprint an article, contact the Intellectual Property Rights Office at copyrights@ieee.org.

IEEE Software

Editorial: All submissions are subject to editing for clarity, style, and space. Unless otherwise stated, bylined articles and departments, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *IEEE Software* does not necessarily constitute endorsement by IEEE or the IEEE Computer Society.

To Submit: Access the IEEE Computer Society's Web-based system, ScholarOne, at http://mc.manuscriptcentral.com/sw-cs. Be sure to select the right manuscript type when submitting. Articles must be original and not exceed 4,700 words including figures and tables, which count for 200 words each.

IEEE prohibits discrimination, harassment and bullying: For more information, visit www.ieee.org/web/aboutus/whatis/policies/p9-26.html.

Printed with inks containing soy and/or vegetable oils

SUSTAINABLE FORESTRY INITIATIVE Certified Chain of Custody At Least 25% Certified Forest Content www.sfiprogram.org SFI-01042

INSIGHTS: **SHARING EXPERIENCE**

Seeking Your Insights

Cesare Pautasso and Olaf Zimmermann

HOW CAN WE keep knowledge from evaporating? Wouldn't it be nice if your valuable experience earned in one project could be exchanged with everyone? What you learned the hard way would become easily accessible to others (and vice versa). Short stories and longer experience reports, reflections and retrospectives, as well as patterns and styles all attempt to gather useful reusable knowledge nuggets that collectively make up the state of the software practice.

Making a good decision is hard. Personal experience can be a good source of evidence to back your decisions, but this is usually limited because you can't experience everything yourself. So, you rely on trusted sources of experience—for example, a colleague, another member of your professional network, or a well-known authority in the field. Additionally, you can attend a reputable conference, search through experience-sharing sites (for example, InfoQ, www.infoq.com, and Stack Overflow, http://stackoverflow.com), and even read magazines (as you're doing now). Written sources have the advantage that the shared experience is "aged." Writing stuff down forces you to reflect on your decision—successful sharing implies hardening—so publishing written knowledge is beneficial for both sides.

This Insights column is one place to write up knowledge nuggets. We're grateful to *IEEE Software* for the opportunity to continue along the path that Linda Rising paved to give a voice to busy software professionals and let their stories be heard. This column's goal remains unchanged—share real-world experience and take a snapshot of where practical software engineering has been, is now, and is heading.

The *IEEE Software* legacy includes foundational and influential articles such as "Reverse Engineering and Design Recovery: A Taxonomy," "Visualizing the Performance of Parallel Programs," "Who Needs an Architect?," "The 4+1 View of Architecture," and "Global Software Development," and other frequently cited ones.[1] Articles in a magazine such as *IEEE Software* are peer reviewed and professionally edited. They can be viewed as a trusted, curated source of experience. And, like conferences, they might provide just enough serendipity so that you can find insights into topics you normally wouldn't look into.

What We're Looking For

The knowledge we're looking for falls into these broad categories:

- patterns of all kinds,
- contemporary architectural styles,
- proven methods and techniques, and
- emerging technologies and tools.

We're interested in hearing about both your positive and negative experiences applying these categories in a given context with reproducible effects. We greatly appreciate critiques, actionable comments, and constructive advice.

Your contributions should be more mature and refined than what you would normally find on most blogs, but just as focused, timely, and relevant. The advantage? Your insights will be presented to the broad *IEEE Software* readership and preserved as part of the IEEE Xplore digital library (http://ieeexplore.ieee.org).

Getting Started

To help you get started, the following questions are intended to provoke a reaction that should lead you to the insights we seek. We ask these questions regularly when performing software reviews. Also, insightful answers to these questions in the context of similar projects or from trusted references have helped us when we worked on industry projects.

We've structured the questions roughly following the basic software engineering life cycle. You don't need to answer them all—it's okay if you pick a few, as long as your answers are substantial enough to be relevant to our readers.

0740-7459/15/$31.00 © 2015 IEEE

INSIGHTS: **SHARING EXPERIENCE**

THE CONTEXT OCTOPUS

Philippe Kruchten's octopus-and-frog metaphor lists eight dimensions of context: (system) size, (system) criticality, system age, team distribution, rate of change, preexistence of a stable architecture, governance (including management rules), and business model (internal system, commercial product, or open source software).[1] These dimensions are worth knowing and disclosing when it comes to experience sharing—one-size-fits-all doesn't work in software engineering.

Reference

1. P. Kruchten, "Contextualizing Agile Software Development," *J. Software Evolution and Process*, vol. 25, no. 4, 2013, pp. 351–361.

Context and Problem

What kind of software project are you describing? In what context did it take place, in terms of the eight dimensions of Philippe Kruchten's context octopus (see the sidebar)?

What problem were you trying to solve? How closely did you involve your problem's stakeholders in the project? How did you deal with their feedback and changing requirements?

Project Management

How did you estimate, manage, and mitigate the technical risk? How much technical debt did you accumulate, and how did you deal with it?

Did you follow some particular software engineering method, set of methods, or practices? For example, would you consider your project agile?

Architecture Design

What were your three most relevant architectural decisions? How did you find and evaluate design alternatives (solution options) for them? How did you pick one? Are you still content with your decisions?

What's your experience with particular modeling notations in real-world projects? Which parts of the system did you model and why? Did this pay off?

Development and Test

How did you test that functional and nonfunctional requirements were satisfied? Did you also try to prove this?

How long was your release cycle? How was your experience with continuous integration, test automation, and software configuration management (versioning)?

Operations and Maintenance

What was your approach to IT service or systems management? Did the chosen frameworks, libraries, and tools deliver on their promises?

If you applied a DevOps (development operations) approach, how did you benefit from it in the short and long terms?

Reflection

When did the constructed system go live? How did it live up to your expectations? How did it survive against the actual workload, and how did you evolve it in response to a growing workload?

In retrospect, what went well and what didn't? Could you solve all the problems (on time and within budget)? What will you do differently next time?

Examples of Insights

Here are some examples of insights we're looking for.

In software development, as in real estate, location matters.[2] Meaningful communication between team members decreases as distance increases. This holds for teams working with people located around the world but also affects the office layout of colocated teams.

Large organizations have difficulty enforcing compliance with technical standards because top-down communication might not be sufficient to get everyone on board.[3] This isn't necessarily a problem of convincing people to commit, but of simply making them aware of the decisions affecting them. As opposed to relying on centralized document archives, which must be continuously searched for relevant information, it helps to push the knowledge directly to the intended recipients when they need it to perform their tasks.

Software architects can learn something from meteorologists.[4] In the same way it's important to forecast an incoming hurricane's path, agile software architects need to anticipate future changes in their design and forecast how the software system will likely evolve.

Given the shortage of skilled IT personnel, it might pay to look for talent outside traditional pools.[5] Not only electronics engineers or graduates in math and physics, but also people with a creative-arts background might show the out-of-the-box thinking and problem-solving

skills a software development position requires. Additionally, the latter folk might be more motivated to keep sharpening their skills on the job.

When you're migrating a legacy system to a new architecture—for example, to turn it into a cloud native application—it helps to have the migration team include the following two roles.[6] The *system steward* knows about the existing system and ensures that the pace of the migration doesn't break it. In contrast, the *future visionary* is completely immersed in the latest technologies and helps push the project toward the endgame by stretching the team's knowledge and abilities.

These are all concrete, actionable pieces of knowledge you can share without compromising your organization's intellectual property. Such knowledge pieces emerge from all industry sectors and business domains and might cover one or more software engineering life-cycle phases. It should be possible to try out your insights and experiment with them in a new project or a well-established one that seeks some improvement. What we're not looking for are buzzword-filled marketing pitches or short-lived, product-specific experiences.

To get your insights presented to our readers, we offer an open ear and a patient hand to coach you and shepherd your drafts. Between us, we share more than 30 years of academic and industry experience in collecting, shaping, and revising technical prose and advising, guiding, and mentoring authors. We're committed to achieve a quick turnaround with feedback about proposals with your initial ideas. You'll also get professional support for editing your story as it goes through the publishing process.

Send in your best insights. We'll be pleased if you can convince your friends and colleagues to share their stories, experience, and knowledge with us and everyone else.

ABOUT THE AUTHORS

CESARE PAUTASSO is an associate professor of informatics at the University of Lugano. His research group focuses on building experimental systems to explore the architecture, design, and engineering of next-generation Web information systems. Previously, he was a researcher at the IBM Zurich Research Lab and a senior researcher at ETH Zurich. His teaching, training, and consulting activities, spanning both industry and academia, cover advanced topics related to emerging Web technologies, RESTful business process management, and cloud computing. He's a coauthor of *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST* (Prentice Hall, 2012). Pautasso received a PhD in computer science from ETH Zurich. He was the program cochair of ICSOC (Int'l Conf. on Service-Oriented Computing) 2013, ECOWS (European Conf. on Web Services) 2010, and Software Composition 2008. He also initiated the Workshop on RESTful Design (WS-REST) at the WWW conference. He's an advisory board member of EnterpriseWeb and a senior member of IEEE. Contact him at c.pautasso@ieee.org, and follow him @pautasso.

OLAF ZIMMERMANN is a professor of software architecture and an institute partner at the University of Applied Sciences of Eastern Switzerland in Rapperswil, Switzerland. Previously, he spent 20 years in industrial research and development and in professional services. He's particularly interested in architectural decision making and design knowledge sharing. As a senior certified IT architect, he has contributed to many company-internal knowledge management initiatives (both successful ones and less successful ones). He's the author of practitioner articles and scientific papers (including award-winning ones), and a contributor to textbooks. Zimmerman received a PhD in computer science from the University of Stuttgart. He has been on the organizing committees of the OOPSLA (Object-Oriented Programming, Systems, Languages, and Applications) conference, the SATURN (Software Engineering Institute Architecture Technology User Network) conference, and WICSA (Working IEEE/IFIP Conf. on Software Architecture) and has been on the *IEEE Software* advisory board since 2011. Contact him at ozimmerm@hsr.ch or www.ozimmer.de.

References

1. D. O'Leary, "The Most Cited *IEEE Software* Articles," *IEEE Software*, vol. 26, no. 1, 2009, pp. 12–14.
2. L. Rising, "Why Can't We All Play Nice?," *IEEE Software*, vol. 29, no. 5, 2012, pp. 7–10.
3. H. Wesenberg and E. Landre, "Making Architecture Matter," *IEEE Software*, vol. 29, no. 3, 2012, pp. 21–23.
4. E. Richardson, "What an Agile Architect Can Learn from a Hurricane Meteorologist," *IEEE Software*, vol. 28, no. 6, 2011, pp. 9–12.
5. W.A. Risi, "Next-Generation Architects for a Harsh Business World," *IEEE Software*, vol. 29, no. 2, 2012, pp. 9–12.
6. J. Crabb, "The BestBuy.com Cloud Architecture," *IEEE Software*, vol. 31, no. 2, 2014, pp. 91–96.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.

Editor: Gerard J. Holzmann
NASA/JPL
gholzmann@acm.org

Code Inflation

Gerard J. Holzmann

MOST PEOPLE DON'T get too excited about software. To them, software applications are like cars: inconspicuous when they work, and merely annoying when they don't. Clearly, cars have been getting bigger and safer over the years, but what about software? It sometimes seems as if it has just gotten bigger, not safer. Why?

> Software tends to grow over time, whether or not there's a need for it.

If you compare the state of today's software development tools with those used in, say, the '60s, you of course see many signs of improvement. Compilers are faster and better, we have powerful new integrated program development environments, and there are many effective static-source-code-analysis and logic-model-checking tools that help us catch bugs. This would have made a fabulous difference if our software applications still looked like they did in the '60s. But they don't.

Many of my NASA colleagues are astronomers or cosmologists. To explain how rapidly things are changing in software development, I've often been tempted to make an analogy with their field. One of the first things you learn in cosmology is the theory of inflation. The details don't matter too much here, but in a nutshell, this theory postulates that the universe started expanding exponentially fast in the first few moments after the Big Bang and continues to expand. The parallel with software development is easily made.

The First Law

Software too can grow exponentially fast, especially after an initial prototype is created. For example, each Mars lander that NASA launched in the past four decades used more code than all the missions before it combined. We can see the same effect in just about every other application domain. Software tends to grow over time, whether or not a rational need for it exists. We can call this the "first law of software development."

The history of the `true` command in Unix and Unix-based systems provides a remarkable example of this phenomenon. Shell scripts often employ this simple command to enable or disable code fragments or to build unconditional `while` loops—for instance, to perform a sequence of random tests:

```
while true
do ./test `rand`
done
```

The `/bin/true` and `/bin/false` commands first appeared in January 1979 in the seventh edition of the Unix distribution from Bell Labs. They were defined as tiny command scripts:

```
$ ls –l /bin/true /bin/false
-rwxr-xr-x 1 root root 0 Jan 10 1979 /bin/true
-rwxr-xr-x 1 root root 7 Jan 10 1979 /bin/false
```

Yes, `true` was actually defined fully with an empty file. How did it work?

Because `true` contained nothing to execute, it always

 0740-7459/15/$31.00 © 2015 IEEE

IEEE Software | Previous Page | Contents | Zoom in | Zoom out | Front Cover | Search Issue | Next Page | Qmags THE WORLD'S NEWSSTAND®

completed successfully, returning the success value of zero to the user. The false command contained seven characters (including the line feed at the end), to return a nonzero value, which signified failure:

```
$ cat /bin/false
exit 1
```

This implementation would seem to leave nothing left to desire, but that would contradict the first law of software development.

In the first commercial version of Unix from 1982, marketed as System III, the implementation of false changed from exit 1 to exit 255, for unclear reasons, but taking up two more bytes. Then, in a version created for the PDP-11 microcomputer in 1983, the implementation of true grew to 18 bytes, and the empty file now contained a comment:

```
@(#)true.sh 1.2
```

In a 1984 version of Unix, things started heating up, and true grew to 276 bytes. The contents were now a boilerplate AT&T copyright notice claiming intellectual ownership of the otherwise still empty file.

A 2010 Solaris distribution further upped the ante by replacing the shell script with a 1,123-byte C source program consisting of a main procedure that called the function _exit(0). The C program for false similarly had main call _exit(255). Both programs also contained a hefty new copyright notice. If I compile these programs on my system today, the executables tap in at 8,377 bytes each.

We're not done yet. The executable for the most recent version of true on my Ubuntu system is no fewer than 22,896 bytes:

```
$ ls –l /bin/true /bin/false
-rwxr-xr-x 1 root root 22896 Nov 19 2012 /bin/true
-rwxr-xr-x 1 root root 22896 Nov 19 2012 /bin/false
```

The source code for this command has grown to 2,367 bytes and includes four header files, one of which is itself 16 Kbytes of text. That's quite a change from the zero bytes in 1979, and all that without any significant difference in functionality.

If you're still on the fence with this: no, true really doesn't need a –version option to explain which version of the truth the command currently represents. Nor does it need a –help option, whose only purpose seems to be to explain the unneeded –version option. And just in case you were thinking about this: true and false also don't need an option that can invert the result, or one that would let these commands send their result by email to a party of your choice. Some have joked that all software applications continue to grow until they can read and send email. This hasn't happened with the two simplest commands in the Unix toolbox just yet, but we seem to have gotten close.

Table 1 shows how the source code and executable code for true have grown. Figure 1 graphs the executable program's growth. The *y*-axis is a log scale so that

TABLE 1

The growth of the source code and executable code of the Unix true command.

Year	Source code size (LOC)	Executable size (LOC)
1979	0	0
1983	18	18
1984	276	276
2010	1,123	8,377
2012	2,367	22,896

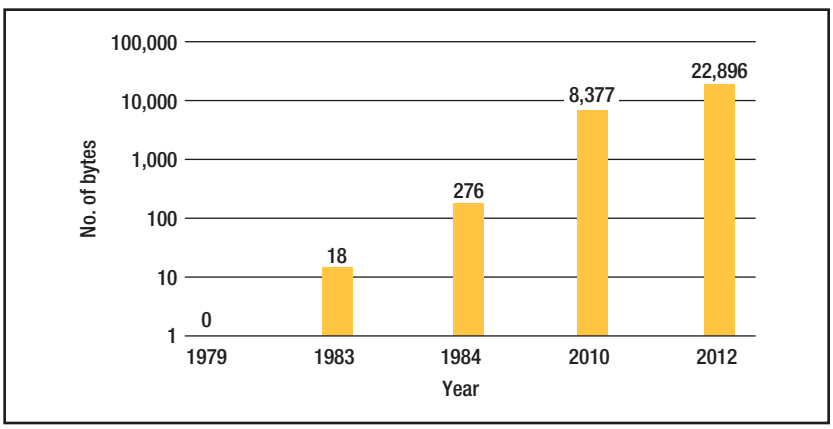


FIGURE 1. The size of /bin/true over time. The *y*-axis is a log scale so that the early numbers aren't completely drowned out by the later ones.

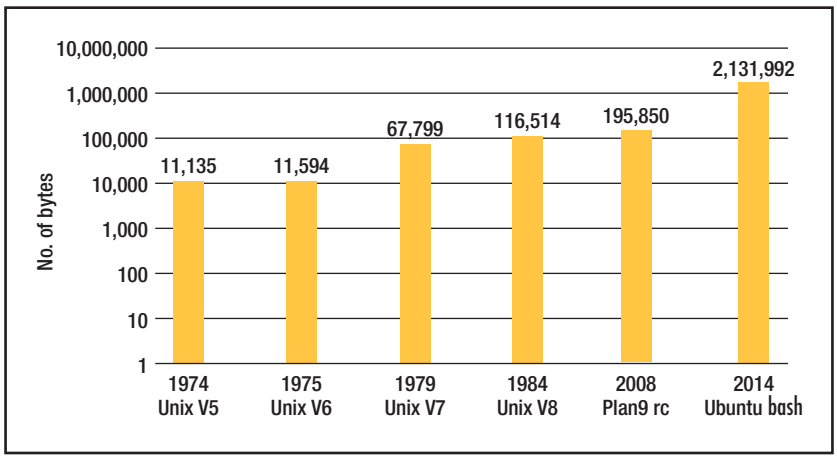


FIGURE 2. The source code size over time for the default command shell on Unix and Unix-like systems. From approximately 11 Kbytes in fifth-edition Unix in 1974 to 2.1 Mbytes for the **bash** shell 40 years later is an increase of 191 times.

the early numbers aren't completely drowned out by the later ones.

Just like in the theory of inflation, the implementation of **/bin/true** increased infinitely fast in the first few years since it was created (because, like the universe, it started at a size of zero). Okay, we're not talking 10^{-32} seconds; we're moving more at humanly achievable speeds here. Once we got to a nonzero size, the expansion continued steadily, with the size increasing more than three orders of magnitude since 1983. (You can find more about the curious history of the **/bin/true** command at John Chambers blog, http://trillian.mit.edu/~jc/;-)/ATT_Copyright_true.html. An online archive of many early Unix source code distributions is at http://minnie.tuhs.org/cgi-bin/utree.pl.)

The best part of all this is perhaps that the copies of **true** and **false** in your system's **/bin** directory are no longer the ones that actually execute when you use these commands in a shell script. Most command shells today define these two commands as built-ins and bypass the externally defined versions. You can check this with the **bash** shell, for instance, by typing **type true** at the command prompt. On most systems, the answer will be **true is a shell builtin**.

If such code inflation can happen to code that's this trivial, and in some ways even redundant, what happens with code that's actually useful? I already mentioned that later versions of the default command shell on Unix and Unix-like systems picked up additional functionality with the interception of calls to **true** and **false**.

Figure 2 shows how the source code for the shell itself, measured in raw bytes, has grown, again using a log scale for the *y*-axis. From approximately 11 Kbytes in fifth-edition Unix in 1974 to 2.1 Mbytes for **bash** 40 years later is an increase of 191 times. Pick almost any other software application, from any domain, and you'll see the same effect.

cat –v

In the early days of Unix development, an attempt was made to reduce the number of command-line options of all standard applications. The thinking was that if additional command-line options were needed, the original code for an application probably wasn't thought out carefully enough. In 1983 at the Usenix Summer Conference, Rob Pike gave an often-quoted presentation on this topic called "Unix Style, or cat –v Considered Harmful." (For more on the presentation, visit http://harmful.cat-v.org/cat-v.) Rob noticed with some dismay that the number of options for the original **cat** command had increased from zero to four. That didn't help. If you check your system today, you'll see that the number of options for this same basic command has reached 12, with seven additional options that you can use as aliases to the others.

So, why does software grow? The answer seems to be: because it can. When memory was measured in Kbytes, it simply wasn't possible to write a program that consumed more than a fraction of that amount. With memory sizes now reaching Gbytes, we seem to have no incentive to pay attention to a program's size, so we don't.

Does it matter? Clearly, it doesn't matter much for the implementation of **true** or **false**, other than that we might object on philosophical grounds. But for code that matters, it might well make a difference. This brings us to the next two laws of software development: all nontrivial code has defects, and the probability of nontrivial defects increases with code size. The more code you use to solve a problem, the harder it gets for someone else to understand what you did and to maintain your code when you have moved on to write still larger programs.

Dark Code

Large, complex code almost always contains ominous fragments of "dark code." Nobody fully understands this code, and it has no discernable purpose; however, it's somehow needed for the application to function as intended. You don't want to touch it, so you tend to work around it.

The reverse of dark code also exists. An application can have functionality that's hard to trace back to actual code: the application somehow can do things nobody programmed it to do. To push the analogy with cosmology a little further, we could say that such code has "dark energy." It provides unexplained functionality that doesn't seem to originate in the code itself. For example, try to find where in the current 2.1 Mbytes of Ubuntu source code for the `bash` shell the built-in commands `true` and `false` are processed. It's harder than you might think.

Software development has one important difference from astronomy or cosmology. In our universe, we can do more than just watch and theorize: we can actually build our universe in the way we think will perform most reliably. Astronomers can't do much about the expansion of the universe other than study it. But in software development we can, at least in principle, resist the temptation to continue to grow the size of applications when there's no real need for it.

So now it's your turn. Instead of just adding more features to the next version of your code, resolve to simplify it. See if you can make the next release smaller than the last one. To get started, if you work on a Linux system, take a stand and replace the gargantuan modern version of `/bin/true` with the original empty executable file. Similarly, replace that newfangled version of `/bin/false` with the single line `exit 1`, which works just as well. You'll feel better, and you'll save some disk space. As the writer Antoine de Saint Exupéry famously noted, "Perfection is achieved not when there is nothing more to add, but when there is nothing more to remove."

GERARD J. HOLZMANN works at the Jet Propulsion Laboratory on developing stronger methods for software analysis, code review, and testing. Contact him at gholzmann@acm.org.

REQUIREMENTS

Editor: **Jane Cleland-Huang**
DePaul University,
jhuang@cs.depaul.edu

AUDIO

Injecting Value-Thinking into Prioritization Decisions

Jane Cleland-Huang

MOST PROJECTS have more potential requirements than they have resources to deliver them. Regardless of whether you're building software to control automobiles, manage patient healthcare records, or calculate driving directions, you need to think long and hard about which features to include in your product and how to prioritize and sequence their delivery. Your decisions will likely have major ramifications on your product's success and marketability.

People have proposed many approaches for selecting and prioritizing features. The most popular ones are based on various triaging schemes for classifying requirements into high, medium, or low priorities or on assigning points to different features and using the accumulated points to rank the features. However, these techniques are naive and easily influenced by the composition of the stakeholders in the room. Better approaches seek consensus—through bringing together key stakeholders, identifying tradeoffs, articulating value propositions, and reaching agreement. One way to do this is through the approach I describe here, which takes into account the features' value.

> It's a bit naive to believe that a simple rule will always produce the winning solution.

Story Mapping and Optimizing Value

Jeff Patton's latest book describes *story mapping*, an agile practice.[1] Story mapping addresses many problems inherent in a traditional backlog and exposes prioritization decisions to the team as a whole. A visual workspace replaces the backlog's flat list. Essential stories form a horizontal backbone along a timeline. Stories detailing each step, or simply occurring around the same time, are added vertically beneath the essential stories. Constructing a story map engages project stakeholders in actively planning releases and thrashing out the delivery sequence. Figure 1 shows a simple story map.

Agile release planning is often driven by the goal of identifying and delivering a *minimal viable product* (MVP).[2] Steve Blank and Eric Reis coined this term to describe the deliverable that maximizes feedback from hands-on users at the lowest risk. This makes a lot of sense. Instead of prioritizing user stories by module, you identify a minimal slice of user stories that cut across the system and that, when deployed, bring real value to the customer. In my own agile experiences, we've worked exactly this way. Our team has met, thrashed out ideas, and identified an initial product we can place into our users' hands.

However, release planning has another aspect. Instead of focusing only on validated learning, you should also consider the financial impact of the features you deliver. The book *Software by Numbers*, which Mark Denne and I wrote over a decade ago, introduced *incremental funding* and showed how to sequence *minimum marketable features* (MMFs) so as to optimize project value.[3] MMFs are the smallest unit of func-

0740-7459/15/$31.00 © 2015 IEEE

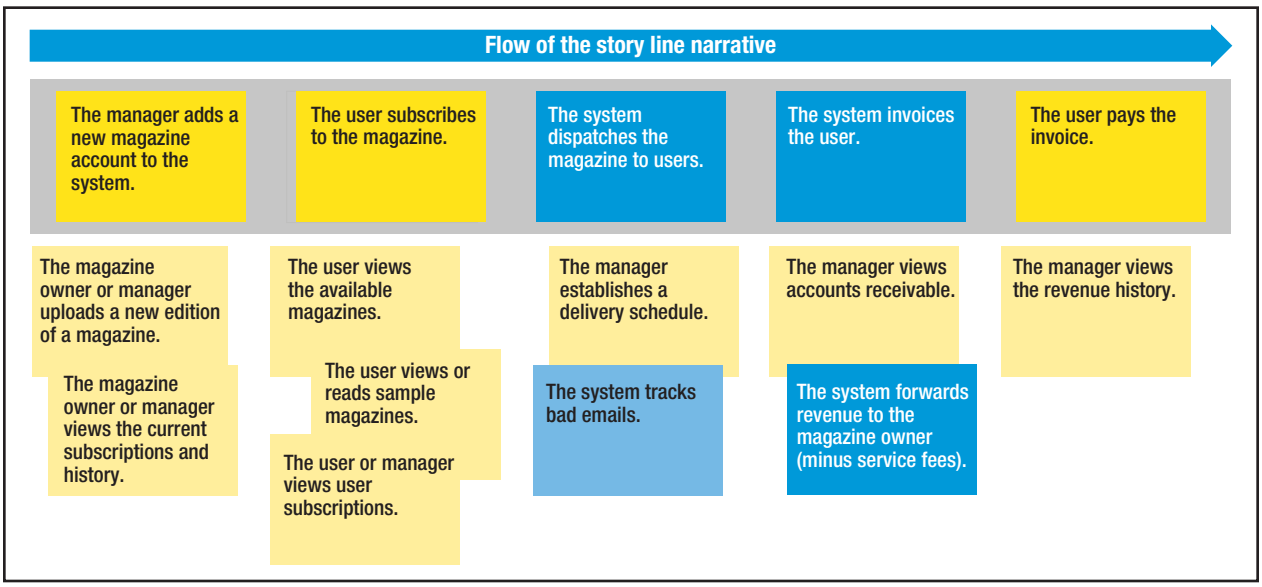


FIGURE 1. A simple story map for Mags-R-Us, a hypothetical business that acts as the middleman for online magazine delivery. Gray indicates the project backbone, yellow indicates user actions, and blue indicates system actions.

tionality that delivers something of value to the customer. Denne and I measured value in terms of revenue, reduced operating costs, and intangibles such as increased customer loyalty. We showed that smart delivery sequences enabled early revenue that can fund the remainder of the project.

Given multiple objectives, it isn't always obvious which parts of the system to build first. The goal is to identify a set of features that maximize validated learning potential while returning real value to the customer. To complicate issues, necessary architectural decisions often impact the timeline. For example, you might need to decide whether to build the simplest possible design first and then refactor later to support more functionality, or build the necessary infrastructure early in the project.

For sure, you could follow the agile mantra and always build the simplest solution first. However, it's a bit naive to believe that a simple rule will always produce the winning solution. The mantra creates a rule of thumb to be followed regardless of whether it's the best decision for a particular project. This also somewhat contradicts the notion that agile team members should be empowered to make their own informed decisions.

The approach I now describe injects value-thinking into prioritization. These ideas stem from my earlier work on incremental funding; here, I present them in a lightweight format and integrate them into story mapping.

A Simple Example

Imagine you're starting Mags-R-Us, a company that acts as the middleman for online magazine delivery. You need to develop a software application to support your business model. You gather together some project stakeholders and brainstorm a set of user stories. These include sign-up, magazine management, dispatch, invoicing, and payments. Your business plan is to build your customer base and iron out problems in your system by launching the first magazine without charging a fee, in order to build a customer base. However, you plan to start charging fees within the first few months. Furthermore, several potential magazines are already lined up to use your delivery service.

You organize your stories into the story map in Figure 1. Yellow indicates user actions; blue indicates system actions. The story map starts when a manager adds a magazine to the system. New editions of the magazine are regularly uploaded. Users can subscribe to the magazine, and the system dispatches it on its planned release dates to all subscribers. The system then invoices the subscribers (perhaps annually), who pay the invoice. In this example, the story map isn't equivalent to a use case because it captures a more general life-cycle narrative.

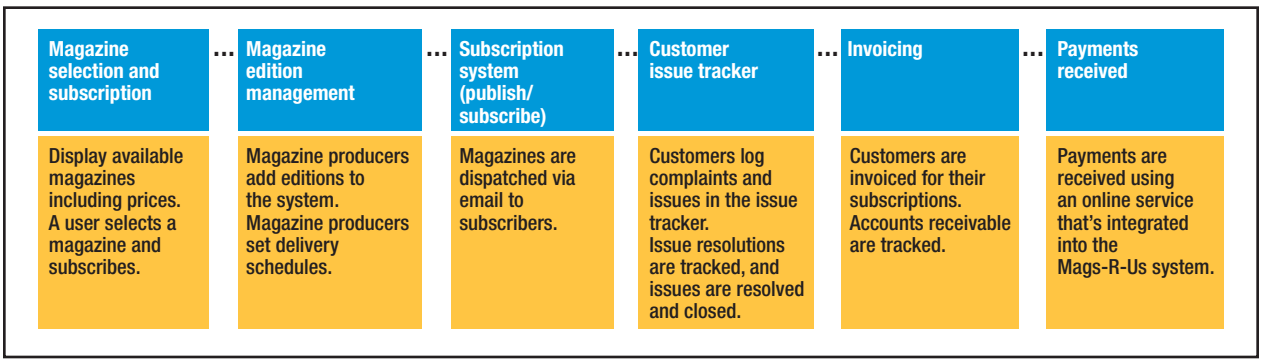


FIGURE 2. In this delivery sequence, called the "big-bang approach," the minimal viable product (MVP) contains all the essential functionality to support the Mags-R-Us business.

TABLE 1

The return on investment (ROI) for the big-bang approach, over time.*

Delivery sequence	Development period												Net
	1	2	3	4	5	6	7	8	9	10	11	12	
Magazine selection (browser)	–5	0	0	0	0	0	0	0	0	0	0	0	–5
Magazine edition manager (browser)		–5	0	0	0	0	0	0	0	0	0	0	–5
Subscription infrastructure			–10	0	0	0	0	0	0	0	0	0	–10
Customer issue tracker				–5	0	0	0	0	0	0	0	0	–5
Invoicing					–10	3	4	5	6	7	8	9	32
Payments received						–5	2	2	2	2	2	2	7

** Each figure represents US $1,000.*

The story map serves several purposes. It helps you recognize and write several important user stories, identify the critical ones, and place them in the project backbone (in gray in Figure 1). But you still need to make prioritization decisions. These decisions don't change the backbone. However, they'll influence the order in which you build and release the features and their supporting infrastructure.

Consider two delivery sequences; Figure 2 shows the first one, which I call the "big-bang approach." The first release bundles all the essential functionality to support the Mags-R-Us business. Although this release might prioritize user stories into iterations, the Mags-R-Us system won't come online until all functionality is delivered.

A simple financial analysis can help determine how choosing this sequence affects the value proposition (see Tables 1 and 2). I make a few simplifying assumptions for the purposes of this illustration. First, most primary user stories (see the first column of Table 1) take one iteration to develop. So, the development period is depicted by the negative dollar amount. Also, revenue can be generated only when invoicing features come online; it will increase once online payments are possible. The cash flow becomes positive only in period 11, taking into consideration the net present value (computed at a discount of 2.5 percent per period).

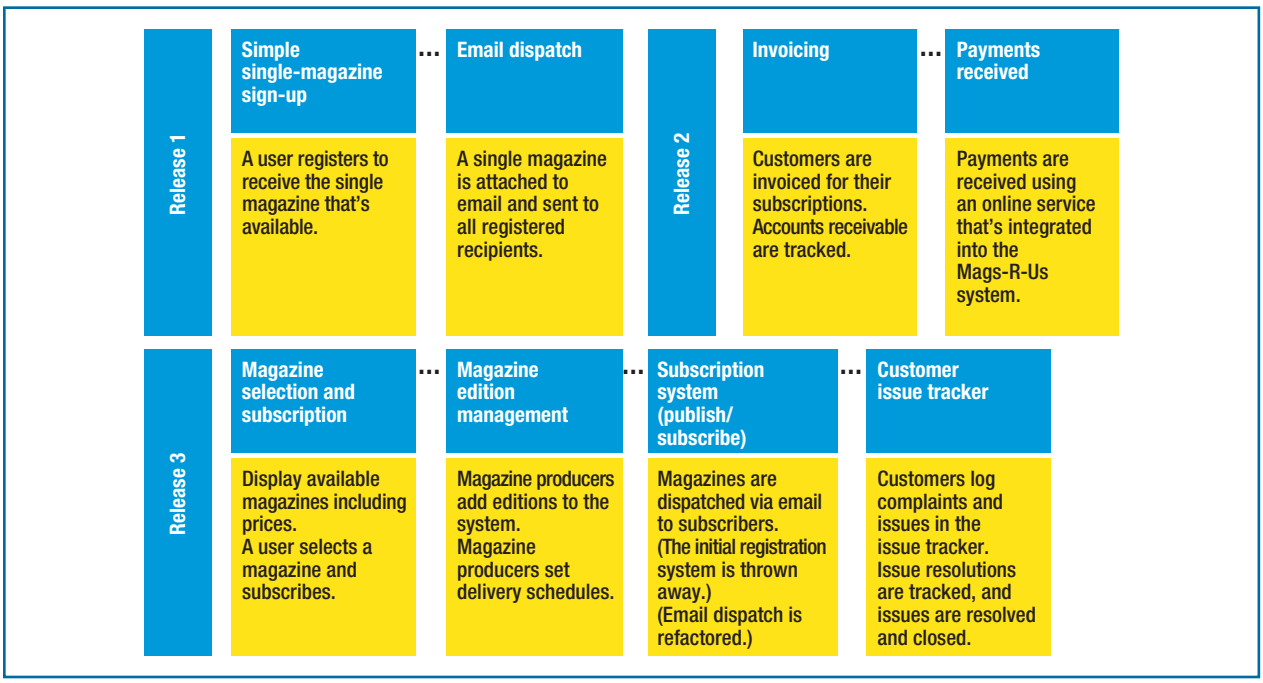


FIGURE 3. This sequence incrementally delivers the MVP in three parts. In this scenario, it delivers greater value than the first sequence (see Figure 2).

TABLE 2

Cash analysis of the ROI for the big-bang approach.*

	Development period											
	1	2	3	4	5	6	7	8	9	10	11	12
Cash	–5	–5	–10	–5	–10	–2	6	7	8	9	10	11
Net cash	–5	–10	–20	–25	–35	–37	–31	–24	–16	–7	3	14
Present value @ 2.50%	–5	–5	–9	–5	–9	–2	5	6	6	7	8	8
Net present value	–5	–10	–19	–23	–32	–34	–29	–23	–17	–10	–2	6

** Each figure represents US $1,000.*

The second delivery sequence decomposes the MVP into three parts, each delivered in a separate, value-enhancing release (see Figure 3).

Again, a basic financial analysis shows how this sequence plays out (see Tables 3 and 4). Release 1 produces intangible benefits through building the customer base. Revenue is generated at the end of release 2, once invoicing and payments go online. Finally, increased revenue follows release 3 as the full functionality is released. Furthermore, early efforts to grow the customer base generate revenue. This delivery sequence delivers greater value than the first one, despite additional costs for refactoring the dispatch component.

The actual numbers in my example aren't as important as the process itself. If, for example, building the customer base in advance fails to increase the starting revenue after the full functionality goes online, the second delivery sequence will actually lose money. At the very least, examining release decisions' financial impact and playing around with various what-if scenarios highlights sensitivity points in the decision making and leads to more informed decisions.

TABLE 3

The return on investment (ROI) for incremental delivery, over time.*

Delivery sequence	Development period 1	2	3	4	5	6	7	8	9	10	11	12	Net
Simple signup	–2	0	0	0	0	0	0	0	0	0	0	0	–2
Email dispatch	–3	0	0	0	0	0	0	0	0	0	0	0	–3
Invoicing		–10	2	2	2	2	2	2	2	2	2	2	10
Online payments			–5	0	0	0	0	0	0	0	0	0	–5
Magazine selection				–5	0	0	0	0	0	0	0	0	–5
Magazine edition management					–5	0	0	0	0	0	0	0	–5
Subscription system						–10	7	8	9	10	11	12	47
Customer issue tracking							–5	0	0	0	0	0	–5

** Each figure represents US $1,000.*

TABLE 4

Cash analysis of the ROI for incremental delivery.*

	Development period 1	2	3	4	5	6	7	8	9	10	11	12
Cash	–5	–10	–3	–3	–3	–8	4	10	11	12	13	14
Net cash	–5	–15	–18	–21	–24	–32	–28	–18	–7	5	18	32
Present value @ 2.50%	–5	–10	–3	–3	–3	–7	3	8	9	9	10	10
Net present value	–5	–14	–17	–20	–23	–29	–26	–18	–9	0	10	21

** Each figure represents US $1,000.*

In writing this column, I'm reminded of the conversations that took place shortly after the release of *Software by Numbers*. Some folks claimed that software developers could never make revenue predictions, therefore financially driven approaches would never work. These people really missed the point and obviously saw software development as an isolated activity. Although business analysts and software developers might not have the skill set or data points to predict revenue streams, they certainly can engage marketing and business folks in project planning and prioritization. Fortunately, there have been many other conversations with folks who reported significant success as they started injecting value-thinking into their feature prioritization processes. I hope you'll give it a try!

References

1. J. Patton, *User Story Mapping*, O'Reilly, 2014.
2. E. Weiss, *The Lean Startup*, Crown Business Books, 2011.
3. M. Denne and J. Cleland-Huang, *Software by Numbers: Low-Risk, High-Return Development*, Prentice Hall, 2003.

JANE CLELAND-HUANG is a professor of software engineering at DePaul University. Contact her at jhuang@cs.depaul.edu.

See www.computer.org/software-multimedia for multimedia content related to this article.

ON COMPUTING

Editor: **Grady Booch**
IBM, grady@computingthehumanexperience.com

All Watched Over by Machines of Loving Grace

Grady Booch

AUDIO

YOU ENTER A CAVE full of twisty little passages, all alike. You and your friends follow the path of a more experienced spelunker, who happens to be a bit portly. He gets stuck in a narrow corridor, unable to move forward or backward, thus trapping the rest of the party. Unfortunately for you, a stream on your side of the cave is rising. If you do nothing, you and your friends will drown. Fortunately for the guide, his head is on the dry side of the blockage, and he will continue to breathe. You have a stick of dynamite with you.

What should you do?

You could use the dynamite to kill your guide, thereby opening a way out for the rest of the party. This is a rather utilitarian philosophy, best summarized by Spock's catch phrase "The needs of the many outweigh the needs of the few … or the one." Killing your guide maximizes life, which we presume is a factor that's not unreasonable to optimize.

If your ethics instead embrace the point of view that all killing is wrong, then you might choose to do nothing, thereby letting the rising waters take their course. Your inaction might be in harmony with your ethics, but it would mean that you and your friends would die and only your guide would survive.

The Double Effect and Software

The ethical conundrum I've posed comes from a 1967 paper by Phillipa Foot.[1] Phillipa's thought experiment has been recast in modern times as the trolley problem.[2] and reformulated in a number of ways. No matter the variation, the center of this experiment attends to the doctrine of double effect, which explores the issue of whether it's permissible to intentionally carry out a harmful act to bring about a good result.[3] Depending on your moral center—and assuming you choose to act in integrity with that center—you would face the question of killing another human to save yourself.

At first glance, this might seem like nothing more than a topic for a late-night philosophical party conversation with friends, fueled by good food and strong drink.

But let's recast the question to make it more interesting.

What would a software-intensive system do? Or more precisely, what would you program a software-intensive system to do, or what would you teach it to do?

A semiautonomous drone will inevitably face this problem: should it terminate the terrorist it has targeted even if an innocent child suddenly enters the kill zone? A semiautonomous car will face a similar choice: if a pedestrian suddenly steps in front of the vehicle, should it swerve, knowing it will hit the car beside it, possibly seriously injuring multiple occupants?

0740-7459/15/$31.00 © 2015 IEEE

Clearly, any discussion of a software-intensive system that actively takes a human life is an emotional subject, so let's dial back the scenario to something that's pure emotion, and reconsider the question.

> Algorithms are essentially thoughtless.

Inadvertent Algorithmic Cruelty

Facebook. Ah, love it or hate it, it's undeniably the way that a billion or so people around the world connect. As an engineer, I respect the mantra that if it works, it's useful. On one hand, I love Facebook for the ability to stay in touch with true and intimate friends (the kind who will show up on your doorstep at 2 a.m. if you call them in need) as well as more casual ones (the kind for whom you will call the police if they show up on your doorstep at 2 a.m. but are still amused in following their media-documented journey through life). On the other hand, I detest it for the way its algorithms take away my control of what I want to see when I want to see it. I am at peace that I am part of Facebook's product content and so participate in Hobson's choice: I have the degrees of freedom to be a member of Facebook or not, and having made that choice in the affirmative, (begrudgingly) accept the consequences.

This is not to say that many of Facebook's features don't annoy me. From the perspective of best user experience practices, I'd judge the experience to be positively user hostile. Their "Year in Review" app is one of those things I find superficial and therefore ignore, but to some—such as Eric Meyer—its very presence is hurtful. As Eric explains in his blog, Facebook presented his Year in Review with a picture of his daughter—who had died earlier that year.[4] To Eric, this was a demonstration of "inadvertent algorithmic cruelty." Eric states the issue eloquently:

> *Algorithms are essentially thoughtless. They model certain decision flows, but once you run them, no more thought occurs. To call a person "thoughtless" is usually considered a slight, or an outright insult; and yet, we unleash so many literally thoughtless processes on our users, on our lives, on ourselves.*

He goes on to observe that "If I could fix one thing about our industry, just one thing, it would be that: to increase awareness of and consideration for the failure modes, the edge cases, the worst-case scenarios."

To say that algorithms are thoughtless is a reasonable and unemotional statement of fact. They have no moral center; they have no sense of right or wrong; they cannot take responsibility for their consequences. Bits cannot feel. However, we who craft such algorithms are expected to be thoughtful. Where does our responsibility begin, and where does it end?

More Dilemmas

The artist collective !Mediengruppe Bitnik (the ! is actually part of their name) developed the Random Darknet Shopper, a bot programmed to make random purchases of $100 in bitcoins every week in a darknet marketplace (https://wwwwww.bitnik.org/r). Over the past year, their bot has purchased cigarettes, counterfeit branded clothing, master keys, and drugs. Some of these purchases were legal, but many were not.

The good folks at the *Guardian* have asked the right question:

> *Can a robot, or a piece of software, be jailed if it commits a crime? Where does legal culpability lie if code is criminal by design or default? What if a robot buys drugs, weapons, or hacking equipment and has them sent to you, and police intercept the package?*[5]

In the case of the Random Darknet Shopper, Domagoj Smoljo, one of its creators, acknowledged that "We are the legal owner of the drugs—we are responsible for everything the bot does, as we executed the code. But our lawyer and the Swiss constitution says [sic] art in the public interest is allowed to be free."[5]

At least in this case there exists a legal safe harbor. In different legal jurisdictions, it might not be so. For example, Google is facing a case in Hong Kong over its ranking algorithms, which are programmed as well as learned.[6] Albert Yeung is suing Google because it offers up related search terms for his name that point to criminal gangs, a situation Albert considers hurtful to his rep-

utation. As the High Court ruled, "Google 'recombines and aggregates' data through its algorithm and therefore can be legally regarded as a 'publisher,' meaning it may be sued for defamation."[6]

Let's consider one more software-intensive system. Event data recorders are the norm for all new automobiles. Privacy issues concerning use of that data abound—a topic for another column—but what about the case in which a driver willingly releases that data in real time to an insurance company to obtain optimal rates? My insurance company might bump my rates up if it finds me speeding, but what if I was speeding for an extended period because I had a sick child in my car whom I had to take to the hospital, or because I was trying to escape, as a victim of some road rage? The presumptive context of any algorithm would be the law; the real context that any human judge would see would entail compassion.

The dilemma here has been well covered—and certainly not well resolved—in the courts regarding mandatory minimums, wherein judges are given zero degrees of freedom to shape the punishment of a certain kind of crime to its particular context. Here we have already seen the societal implication of judgment without compassion; computing makes it profoundly easy to release algorithms without compassion.

Richard Brautigan, author of *Trout Fishing in America*, once wrote a poem, part of which reads

> *I like to think (it has to be!)*
> *of a cybernetic ecology*
> *where we are free of our labors*
> *and joined back to nature,*
> *returned to our mammal*
> *brothers and sisters,*
> *and all watched over*
> *by machines of loving grace.*[7]

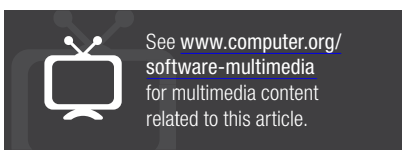
From a theological perspective, judgment is often defined as receiving that which one deserves, mercy as not receiving that which one deserves, and grace as receiving that which one does not deserve. Richard published this poem in 1967, the year Foot proposed her ethical conundrum. In that age, we could only dream of such machines. Now, we build them. Furthermore, these practical ethical conundrums are only beginning to become fully manifest as we slowly and inevitably and irreversibly surrender to these machines of our own creation.

As software professionals, what should we do?

References

1. P. Foote, "The Problem of Abortion and the Doctrine of the Double Effect," *Oxford Rev.*, no. 5, 1967, pp. 5–15; www2.econ.iastate.edu/classes/econ362/hallam/readings/footdoubleeffect.pdf.
2. J. Clark, "How the Trolley Problem Works," HowStuffWorks.com, 3 Dec. 2007; http://people.howstuffworks.com/trolley-problem.htm.
3. "Doctrine of Double Effect," *Stanford Encyclopedia of Philosophy*, 2014; http://plato.stanford.edu/entries/double-effect.
4. E. Meyer, "Inadvertent Algorithmic Cruelty," blog, 24 Dec. 2014; http://meyerweb.com/eric/thoughts/2014/12/24/inadvertent-algorithmic-cruelty.
5. M. Power, "What Happens When a Software Bot Goes on a Darknet Shopping Spree?," *The Guardian*, 5 Dec. 2014; www.theguardian.com/technology/2014/dec/05/software-bot-darknet-shopping-spree-random-shopper.
6. S. Lau, "Hong Kong Tycoon Can Sue Google over 'Autocomplete' Search Suggestions, Court Rules," *South China Morning Post*, 7 Aug. 2014; www.scmp.com/news/hong-kong/article/1567521/hong-kong-court-rules-tycoon-can-sue-google-over-autocomplete-search.
7. R. Bratigan, *All Watched Over by Machines of Loving Grace*, Communication Co., 1967; www.brautigan.net/machines.html.

GRADY BOOCH is an IBM Fellow and one of UML's original authors. He's currently developing *Computing: The Human Experience*, a major transmedia project for public broadcast. Contact him at grady@computingthehumanexperience.com. Follow him on Twitter @grady_booch.



VOICE OF EVIDENCE

Editor: Tore Dybå
SINTEF
tore.dyba@sintef.no

Managing Technical Debt

Insights from Recent Empirical Evidence

Narayan Ramasubbu, Chris F. Kemerer, and C. Jason Woodard

TECHNICAL DEBT refers to maintenance obligations that software teams accumulate as a result of their actions. For example, when programmers take a design shortcut to more quickly roll out functionality critical to business stakeholders, they start accumulating maintenance obligations related to the shortcut. Like financial debt, technical debt incurs interest in the form of additional maintenance costs until the debt is fully paid. Ward Cunningham first used the debt analogy as he reflected on his product development experience using object-oriented programming:

> *Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make this transaction tolerable. The danger occurs when the debt is not repaid. … Entire engineering organizations can be brought to a stand-still under the debt load.*[1]

Cunningham's reflections portray managing technical debt as an optimization problem. This inspired us to undertake research projects to rigorously model technical debt and understand its impact on both software product performance and business strategies. Here, we synthesize the empirical findings of our projects to develop practical insights for managing technical debt.

Recognizing Trade-Offs

An important first step is to recognize the trade-offs involved in managing technical debt. Software teams need to clearly understand both the benefits and costs of technical debt in their specific business environment. For example, a firm pursuing an early-to-market business strategy might choose to incur technical debt to speed up product development and accelerate customer acquisition. But, if this debt isn't repaid in a timely way, its benefits might be eroded by the long-term costs associated with poor product reliability and with the difficulty in meeting customer demands.

Our research shows that software teams willing to accumulate technical debt could speed up functionality deployment in their products by as much as three times.[2] However, they had to contend with a threefold increase in the backlog of unresolved errors and a drop of more than 50 percent in long-term customer satisfaction scores.

Three Dimensions to Optimize Technical Debt

Once software teams recognize the specific trade-offs they face, they can move toward optimizing their debt load by tracking appropriate metrics and instituting policies to manage the level of technical debt within the range they're

0740-7459/15/$31.00 © 2015 IEEE

willing to tolerate. Recent research has outlined how to identify and measure technical debt for a variety of programming languages.[3,4] To translate this data into actionable decisions, software teams should map their technical-debt metrics to three important dimensions (see Figure 1).

The first dimension is customer satisfaction needs, which refer to the extent of demands placed on software teams to keep end users satisfied. A variety of factors could affect this satisfaction, including the end users' intrinsic motivation, the perceived ease of use, and the perceived benefits of use. When those three metrics' values are low, software teams must deploy strong triggers for initiating customer adoption and continued product use. Such triggers could be faster rollout of the demanded functionality or enhanced usability. However, these might sometimes come at the expense of not being able to follow recommended standards focused on long-term design stability. Our research shows that, all other things being equal, advancement by a month of the rollout of an end-user-demanded functionality could increase customer satisfaction by up to 55 percent in the short term.[2]

The second dimension is the extent of software reliability demanded by the business. Our research indicates that technical debt accumulated in enterprise software systems tends to increase the chance of system failures. In examining failures of enterprise systems at 48 Fortune 500 firms, we found that technical debt arising from business logic and data schema customizations that violated vendor-prescribed design standards increased the chance of system failures by as much as 62 percent.[4] The extent to which businesses can tolerate such risk can vary; software teams must carefully consider this when deciding to accept or avoid technical debt.

The third dimension is the probability of technology disruption in a firm's environment. We observed a few cases in which technical-debt-laden teams successfully wrote off their debt by substituting newer technology platforms for older ones for product development.[5] However, adopting new technologies shouldn't be seen as a panacea for technical-debt problems. Such technologies often take time to stabilize and might not immediately be conducive to the high-reliability operations some businesses desire.

FIGURE 1. Software teams should map their technical-debt metrics to these three dimensions.

Eight Scenarios

Figure 2 summarizes our recommendations for the eight scenarios spanning the high and low levels of customer satisfaction needs, reliability needs, and the probability of technology disruption.

When both the reliability needs and the customer satisfaction needs are low (the lower-left box in Figure 2), technical debt isn't a significant concern. Software teams can continue their established product development processes without any special attention toward technical debt. These teams must invest in debt-reducing activities only when product functionality growth becomes saturated and no new technology is available to switch to. Otherwise, the teams can continue to accumulate technical debt without significant worry about long-term impacts. They can simply write off the accumulated technical debt by switching to any new technological platform as soon as it's available.

When the reliability needs are high and the customer satisfaction needs are low (the upper-left box in Figure 2), software teams should avoid technical debt. These teams must shun functionality development based on newly released features of a technological platform until the typical initial wrinkles of the new technology are ironed out. Our investigation of the 10-year evolution of an enterprise software product at 69 large client firms showed that teams that avoided technical debt by choosing to be late adopters of functionality had, on average, about 13 times fewer unresolved errors and about seven times lower bug-fixing effort expenditures than teams who incurred technical debt.[2] However, if the probability of technological disruption is high, software teams can relax the policy of being highly selective late adopters and be open to incurring technical debt, especially if it helps to solve tricky performance bottlenecks. Those teams will still need to invest in debt-reduction activities until the new technology matures and is ready for wider deployment.

Software teams in the remaining scenarios in Figure 2 can be more

FIGURE 2. Recommendations for managing technical debt.

open to accepting technical debt. What vary between these scenarios are the timing and extent of debt-reduction investments.

Our research shows that when both the reliability needs and the customer satisfaction needs are high (the upper-right box in Figure 2), it's optimal for software teams to accumulate technical debt only until their products have "taken off." A takeoff indicates that the product has achieved a critical mass of adopters who will likely use it throughout its lifetime. Because this installed base of customers expect high reliability, software teams should aggressively pay off most of the accumulated technical debt. Such a strategy is particularly apt if the probability of technology disruption is low, because it will enhance the product's longevity. Aggressively pursuing debt reduction will help software teams maintain higher software quality and continue to add functionality to keep end users happy. On the other hand, if technological disruption is more likely, software teams can reduce debt in a more targeted way. This is because the opportunity exists to write off technical debt by switching to the new technology as soon as its performance stabilizes.

Finally, when the reliability needs are low and the customer satisfaction needs are high (the lower-right box in Figure 2), software teams can be open to incurring technical debt to quickly deliver end-user-demanded functionality. Because the reliability needs are low, teams can postpone technical-debt reduction until product functionality growth becomes saturated. Meanwhile, as in other scenarios, if a new technology platform emerges, software teams can write off all the accumulated debt by immediately switching to that platform.

We believe a technical-debt policy must be based both on the business context of a firm and on the technological environment in which the firm operates. Completely avoiding technical debt is prudent only when the probability of technological disruption is low, reliability needs are high, and prolonging an existing product's life is a high priority. We recommend that software teams optimize the timing and extent of technical-debt accumulation and reduction on the basis of the three dimensions described in this article.

Our research also highlights three main areas in which the empirical evidence is suggestive, but significant questions remain unanswered.

First, what other aspects of the business environment should influence a software team's technical-debt policy? Our qualitative case studies explored the role of resource munificence, technical capability, and the ability to transfer debt to other members of a firm's business ecosystem.[5] Unsurprisingly, we found that firms with abundant resources tended to take on less debt and pay it off faster than firms with scarce resources. But this finding is hard to interpret normatively. Perhaps firms with more resources should take on more debt because they can more easily pay it off later

(much as wealthy individuals take on debt to finance higher-yielding investments). We also found that more technically capable software teams tended to be more debt-averse. But again, this doesn't imply that good developers shouldn't take shortcuts for sound business reasons.

The ability to transfer debt raises even subtler issues, akin to the problem of moral hazard in economics. If you can shift your debt burden to someone else (for example, your customers or partners), should you? The answer might depend on the long-term consequences (for example, to your reputation or to your platform's growth).

Second, we distinguish between modular and architectural maintenance. The former results in code changes localized in software modules; the latter results in changes distributed across module boundaries. Our research on enterprise software packages indicates that these two types of maintenance undertaken by client firms to reduce technical debt have different effects on system failures, and cause unintended side effects, such as conflicts with future vendor releases.[4] Further investigation is needed to understand what debt-reduction strategies are the most effective in different client-specific environments.

Finally, how does adding or reducing technical debt affect a software product development project's evolution? We've observed that high-debt projects tend to increase rapidly in functionality, but then reach a plateau, whereas low-debt projects start off more slowly but ultimately achieve higher functionality.[2] However, a software system isn't a balance sheet; technical debt is an evocative metaphor for a more complex underlying reality. Taking on technical debt means incurring future maintenance obligations, but might also mean taking a fundamentally different path in exploring the system's design space. That path might turn out to be a shortcut, as when a technology disruption enables a team to switch to a new platform and effectively write off its debt. But it might also be a dead end, where efforts to restructure a debt-laden platform never succeed.[5]

References

1. W. Cunningham, "The Wycash Portfolio Management System," *Addendum to Proc. Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA 92), 1992, pp. 29–30.
2. N. Ramasubbu and C.F. Kemerer, "Managing Technical Debt in Enterprise Software Packages," *IEEE Trans. Software Eng.*, vol. 40, no. 8, 2014, pp. 758–772.
3. B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the Principal of an Application's Technical Debt," *IEEE Software*, vol. 29, no. 6, 2012, pp. 34–42.
4. N. Ramasubbu and C.F. Kemerer, "Technical Debt and the Reliability of Enterprise Software Systems: A Competing Risks Analysis," 2014; http://ssrn.com/abstract=2523483.
5. C.J. Woodard et al., "Design Capital and Design Moves: The Logic of Digital Business Strategy," *MIS Q.*, vol. 37, no. 2, 2013, pp. 537–564.

NARAYAN RAMASUBBU is an assistant professor of business administration at the University of Pittsburgh. Contact him at narayanr@pitt.edu.

CHRIS F. KEMERER is the David M. Roderick Professor of Information Systems at the University of Pittsburgh. Contact him at ckemerer@katz.pitt.edu.

C. JASON WOODARD is an assistant professor of information systems at Singapore Management University. Contact him at jwoodard@smu.edu.sg.

THE PRAGMATIC ARCHITECT

Editor: **Eoin Woods**
Endava
eoin.woods@artechra.com

Architectural Refactoring

A Task-Centric View on Software Evolution

Olaf Zimmermann

SOFTWARE-INTENSIVE SYSTEMS often must be reengineered—for example, owing to unpredictable business changes and technology innovations. Many reengineering activities affect these systems' software architecture. One popular reengineering practice is code refactoring. Given the success of code refactoring, it's surprising that architectural refactoring (AR) hasn't taken off yet. To help remedy that situation, I show here how AR can serve as an evolution technique that revisits architectural decisions and identifies related design, implementation, and documentation tasks.

Introducing Architectural Refactorings

A refactoring aims to improve a certain quality while preserving others. For example, code refactoring restructures code to make it more maintainable without changing its observable behavior.[1] Code refactorings work on machine-readable entities such as packages, classes, and methods so that they can leverage data structures from compiler constructions such as abstract syntax trees.

ARs deal with architecture documentation and the architecture's manifestation in the code and run-time artifacts. So, a single architectural syntax tree doesn't exist. ARs pertain to

- components and connectors (modeled, sketched, or represented implicitly in code),
- design decision logs (in structured or unstructured text), and
- planning artifacts such as work items in project management tools.

ARs address *architectural smells*, which are suspicions or indications that something in the architecture is no longer adequate under the current requirements and constraints, which might differ from the original ones. An AR, then, is a coordinated set of deliberate architectural activities that remove a particular architectural smell and improve at least one quality attribute without changing the system's scope and functionality. An AR might negatively influence other quality attributes, owing to conflicting requirements and tradeoffs.

In my view, an AR revisits certain architectural decisions[2] and selects alternate solutions to a given set of design problems. A decision's execution leads to related engineering tasks, which fall into these categories:

- Tasks to realize structural changes in a design. Such changes have a larger scope than code refactorings and deal with components, subsystems, and systems of systems (and their interfaces).
- Implementation and configuration tasks in development and operations (depending on the AR's viewpoint).
- Documentation and communication tasks, such as modeling activities, technical-writing assignments, or

0740-7459/15/$31.00 © 2015 IEEE

TABLE 1

A structured representation of an architectural refactoring (AR).

AR name How can the AR be recognized and referenced easily?	De-SQL
Context Where (and under which circumstances) is this AR eligible?	The functional viewpoint and the information viewpoint, at both the conceptual level (database paradigm) and asset level (MySQL versus MongoDB) of abstraction.
Stakeholder concerns Which nonfunctional requirements and constraints (including quality attributes and design forces) does this AR affect?	Flexibility (regarding data model changes), data integrity, and migration time.
Architectural smell When and why should this AR be considered?	It takes too long to migrate existing database content when the data model (database schema) is updated.
Architectural decisions to be revisited Which design problems pertain to this AR, and which design options are currently chosen to resolve them?	• The choice of data-modeling paradigm (the current decision is relational). • The choice of metamodel and query language (the current decision is SQL). • The choice of database management system (the current decision is MySQL).
Evolution outline (solution sketch) Which design options should be chosen now? What does the target solution look like?	• Use a document-oriented database such as MongoDB instead of a relational database such as MySQL. • Redesign transaction management and database administration.
Affected architectural elements Which design model elements must be changed (such as components and connectors, if modeled explicitly)?	• Database tier (including server process and backup and restore facilities). • Data access layer (such as patterns for commands and queries, and connection pools).
Execution tasks How can the AR be applied and validated?	• Design the document layout (the pendant to the machine-readable SQL data definition language). • Write a new data access layer and implement SQL-ish query capabilities in the application. • Decide on transaction boundaries (if any). • Document database administration changes (such as command-line queries, update scripts, and backup procedures). • Compare the old and new solutions according to success criteria (such as migration time and the data access layer's performance).

design workshop preparation and facilitation.

My decision- and task-centric view on ARs complements and extends that of Michael Stal, who published the first catalog of ARs in 2007.[3] To document his ARs, Stal uses a simple pattern format with three sections: context, problem, and general solution idea. His ARs include "Break Dependency Cycles" and "Splitting Subsystems," addressing the architectural smells "dependency cycles" and "inadequate partitioning of functionality."[4]

An Example and a Task-Centric Template

The chief technicians at Doodle.com explained in their blog why they switched from using MySQL to MongoDB after several years of productive use of their collaborative online calendar-scheduling service.[5] In this case, the architectural smell was that migrating large production databases after an SQL schema change (such as adding a column to a table) took too long. The affected quality attributes were the productivity in development and operations as well as the performance and scalability of the database and the data access layer. The cause of the smell's symptoms was that relational database management systems weren't designed for that usage scenario: they could handle it, but not optimally.

The solution was to revisit the architectural decisions regarding the database paradigm, query APIs, and

database provider. The technicians decided to use the document-oriented paradigm, one flavor of schemaless NoSQL, and MongoDB as the document database provider. This change improved migration management at the expense of administration and coding effort—new solutions for data access, transactions, and backup management were required.

The Doodle example qualifies as an AR because it revisits certain architectural decisions to improve a quality attribute but doesn't involve code refactoring. The structured AR representation in Table 1 makes it easy to comprehend and apply this AR to a similar project. This example also proposes an AR documentation template; each row in Table 1 is one template entry. Figure 1 shows the resulting template structure.

When using the template shown in Table 1 to document your own ARs, the AR name should be expressive, such as a metaphor. Unlike pattern names, which are typically nouns, AR names should be verbs, like code-refactoring names. The context section can include information about the abstraction level in a software engineering method or viewpoint in an enterprise architecture management framework. Because the AR describes a design change, two solution sketches can be provided: the design before applying the AR, and the design after applying the AR. Architectural elements form a link to the structural design, which might be modeled explicitly, sketched informally, or represented implicitly in code. The task description might refer to work items in agile planning tools or to software engineering activities. Some execution tasks can be automated (just like the execution of many code refactorings), but not all,

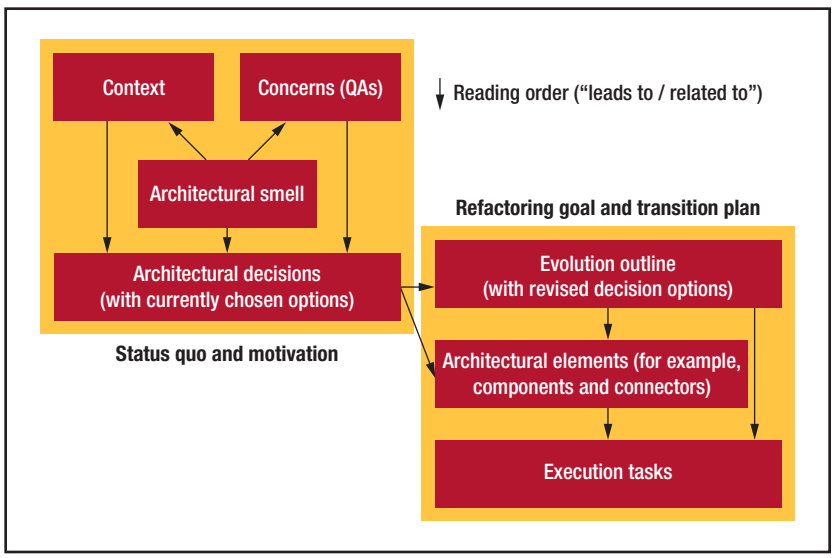


FIGURE 1. The anatomy of an architectural refactoring. The left side presents an architectural smell as observed in a particular context (such as viewpoint or abstraction level), pertaining to certain quality attribute (QA) concerns. The right side sketches the refactored architecture and identifies related design, implementation, and documentation tasks. The architectural decisions to be revisited serve as the glue connecting these two parts.

because ARs operate on a higher abstraction level and larger scale than code refactorings.

Architectural-Refactoring Catalogs

Let's now go broad and look at additional ARs. Table 2 lists basic ARs in two dimensions: architectural viewpoints and types of change. These ARs can be represented as instances of the task-centric template in Figure 1. For example, the tasks for "Introduce Cache" include deciding on a lookup key and cache invalidation strategy, cache distribution, and so on.

In the future, domain- and style-specific AR catalogs might appear, perhaps for financial-services software, game development, or cloud computing. For example, here are three candidate ARs for a prospective catalog for enterprise application modernization:

- Move session state management (for example, from the client or mid-tier server to a database to improve horizontal scaling and to better leverage cloud elasticity).
- Replace scalar parameters with a data transfer object in a service interface contract (to reduce the number of remote calls).
- Streamline a Web client (to reduce the client workload and processing capabilities).

Both basic and specific ARs provide an opportunity for cross-community collaboration between

- Architecture and development. AR execution might involve one

TABLE 2

Viewpoints, types of ARs, and the associated ARs.

Viewpoints	Types of change: Elaboration ARs	Adjustment ARs	Simplification ARs
Functional	Split Component Responsibility	Expose Internal Feature as Component Responsibility	Merge Component Responsibilities
	Shift Responsibility to New Component	Shift Responsibility to Existing Component	Merge Components
	Split Layer (Move Components to New Layer)	Replace Layer	Join Adjacent Layers (Collapse Layers)
Concurrency, Information	Distribute Processing (Introduce Concurrency)	Change Distribution Algorithm (for example, from Round Robin to Priority Driven)	Consolidate Processing (Remove Concurrency)
	Introduce Cache	Change Cache Entry Lookup Key	Remove Cache
	Prepopulate Cache (Load More Eagerly)	Change Cache Cleanup Strategy	Start with Empty Cache (Load Lazier)
Deployment	Assign Logical Component to New Deployment Unit	Change Scaling Strategy (for example, from vertical scale-up to horizontal scale-out)	Merge Deployment Units
	Split Deployment Unit (to deploy on separate existing or new nodes)	Move Deployment Unit (from one server node to another)	Consolidate Nodes
Deployment, Operational	Factor Out Node into New Tier	Split Tier	Collapse Tiers
	Introduce Clustering	Change Load-Balancing and Failover Policy	Remove Clustering

or more code refactorings, which must be stitched together.
- Architecture and project management. AR descriptions organized according to the AR template can serve as planning tasks; the need for AR is an expression of technical debt.
- Architecture and operations (ArchOps). ARs in the deployment viewpoint can serve as a means for communication.

It remains to be seen how to efficiently share and execute ARs. Are templates and catalogs good enough knowledge carriers, or are modeling and collaboration tools more appropriate? Web-based delivery of knowledge has a natural appeal, as Wikipedia demonstrates. Code refactoring started with a book and formal groundwork. Refactoring tools (such as those in Eclipse) were developed much later, after content and theory were established and experience was gained. Any AR tool support would need to tie in with modeling tools supporting UML or architecture description languages, but such support has yet to emerge.

Acknowledgments

I thank Christian Bisig and Mirko Stocker of the University of Applied Sciences of Eastern Switzerland, Rapperswil, for reviewing earlier versions of this article. I also thank the attendees of my OOP 2014 and GI FG SWA 2014 presentations on architectural refactoring for the cloud for their helpful feedback.

References

1. M. Fowler, "Definition of Refactoring," blog, 1 Sept. 2004; http://martinfowler.com/bliki/DefinitionOfRefactoring.html.
2. O. Zimmermann, "Architectural Decisions as Reusable Design Assets," *IEEE Software*, vol. 28, no. 1, 2011, pp. 64–69.
3. M. Stal, "Software Architecture Refactoring," 2007; www.sigs.de/download/oop_08/Stal/20Mi3-4.pdf.
4. M. Stal, "Refactoring Software Architecture," *Agile Software Architecture*, 1st ed., M. Babar, A. Brown, and I. Mistrik, eds., Morgan Kaufmann, 2013.
5. P. Sevinç, "Doodle's Technology Landscape," blog, 14 Apr. 2011; http://en.blog.doodle.com/2011/04/14/doodles-technology-landscape.

OLAF ZIMMERMANN is a professor and institute partner in the Institute for Software at the University of Applied Sciences of Eastern Switzerland in Rapperswil. Contact him at ozimmerm@hsr.ch.

SOFTWARE TECHNOLOGY

Editor: **Christof Ebert**
Vector Consulting Services
christof.ebert@vector.com

Infrastructure as a Service and Cloud Technologies

Nicolás Serrano, Gorka Gallardo, and Josune Hernantes

Software and infrastructure are increasingly consumed from the cloud. This is more flexible and much cheaper than deploying your own infrastructure, especially for smaller organizations. But the cloud obviously bears a lot of risk, as our partner magazine *Computer* emphasized in its October 2014 issue. Performance, reliability, and security are just a few issues to carefully consider before you use the cloud. The perceived savings immediately turn into a large extra cost if your clients and workforce are disconnected for some minutes or critical infrastructure doesn't perform as intended—which happens more often than what brokers and providers disclose. In this issue's column, IT expert Nicolas Serrano and his colleagues examine the enterprise cloud market and technologies. They provide hands-on guidance for making the right decisions. I look forward to hearing from both readers and prospective authors about this column and the technologies you want to know more about. —*Christof Ebert*

CLOUD COMPUTING'S LOW COST, flexibility, and agility are well understood in today's corporate environment. However, to fully exploit cloud technologies, you need to understand their best practices, main players, and limitations.

The concept of cloud computing has existed for 50 years, since the beginning of the Internet.[1] John McCarthy devised the idea of time-sharing in computers as a utility in 1957. Since then, the concept's name has undergone several changes: from service bureau, to application service provider, to the Internet as a service, to cloud computing, and to software-defined datacenters, with each name having different nuances. However, the core concept is the same: providing IT services based on the Internet (the cloud).

The most-used definition of cloud computing belongs to the US National Institute of Standards and Technology (NIST):

> *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*[2]

0740-7459/15/$31.00 © 2015 IEEE

Providers use three well known models (see Figure 1): IaaS (infrastructure as a service), PaaS (platform as a service), and SaaS (software as a service). Here, we focus on IaaS. The next step is to decide on a model for deploying cloud services. In a *public cloud*, a provider provides the infrastructure to any customer. A *private cloud* is offered only to one organization. In a *hybrid cloud*, a company uses a combination of public and private clouds.

To choose the most appropriate cloud-computing model for your organization, you must analyze your IT infrastructure, usage, and needs. To help with this, we present here a picture of cloud computing's current status.

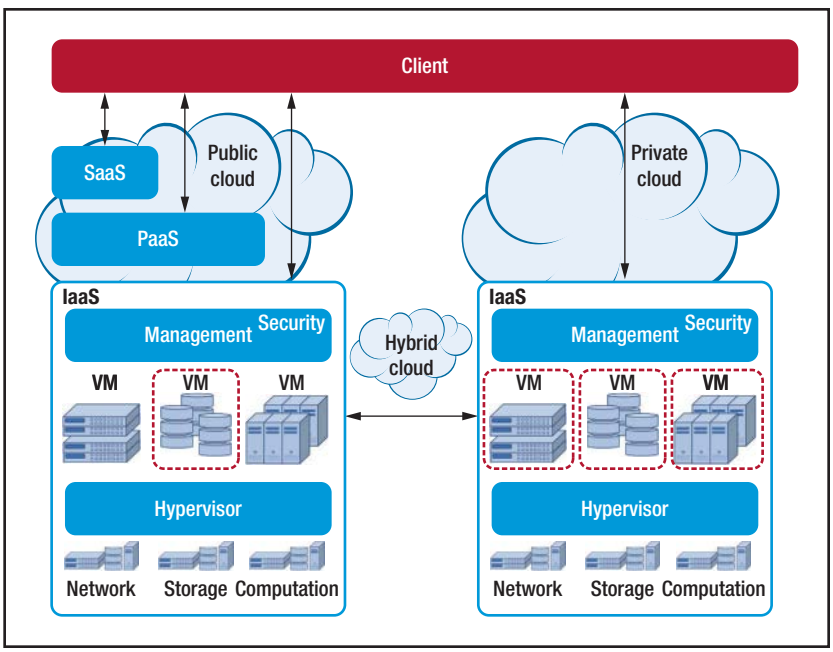


FIGURE 1. The three cloud models. IaaS is infrastructure as a service, SaaS is software as a service, PaaS is platform as a service, and VM stands for a virtual machine.

Cloud Computing Best Practices

As with every new architectural paradigm, it's important to design your systems taking into account the new technology's characteristics. To select a cloud provider or technology, you should understand your requirements in order to list the needed features. Here are some best practices for cloud migration.[3]

An Elastic Architecture

IaaS offers precise scalability. The cloud can outperform physical hardware's classic scale-up or scale-out strategies. To gain as much as you can from this potential, architect your systems and application with as much decoupling as possible, using a service-oriented architecture and using queues between services.

Design for Failure

High scalability has limitations. IaaS technology and architecture lead to a less robust system because you're replacing hardware with several software layers, adding obvious complexity and failure points. Redundancy and fault tolerance are primary design goals.

Besides having an established backup strategy, to assure business continuity, ensure your system is prepared for reboots and relaunches. Automation in your deployment practice is a must, with recipes for server configuration and deployment. Providing automation requires new development practices (development and operations management, continuous integration, test-driven development, and so on) and new tools such as Chef, Puppet, or Ansible.

High Availability

IT resource disruption has a huge negative impact on any business. Lost control of the underlying infrastructure when moving to the cloud, and the fact than the service-level agreement (SLA) won't cover all the incurred costs, should lead you to design with outages and high availability in mind. With the ease of creating virtual instances, deploying clusters of servers or services is a popular approach. In this scenario, load balancing is a well-established technique for operating with clusters; it's an important feature to consider when selecting a cloud provider.

It's also important to use several available zones or at least different datacenters to make your system as robust as possible. Amazon Web Services (AWS) experienced this in April 2011 when its systems didn't run or ran intermittently for four days. Separating clusters into regions and datacenters will increase your resources' resilience.

Performance

You need to consider the technology's limitations regarding performance—

TABLE 1

Cloud provider characteristics.

Company	Reported usage (%)*[4]	Trend 2014–2015 (%)*[4]		Technology	Price (US$, Aug. 2013)[5]	Functionality or features[6]					Datacenters	Vendor positioning[7]
		Experimenting	Plan to use			Advanced features or functionality	Hybrid cloud	Network management	Service-level agreement (%)[4]	Average service time (yrs.)		
Amazon Web Services	54	23	9	Proprietary	66	Good	Yes	Partial	99.95	5+	US East and West Coast, Ireland, Japan, Singapore, Australia, Brazil, China	Leader
Microsoft Azure	6	20	8	Proprietary	65	Good	Yes	Partial (unavailable for Linux)	99.95	4	US, Ireland, Netherlands, Hong Kong, Japan, Singapore, China, Brazil	Leader
Rackspace	12	14	12	OpenStack	116	Adequate	Yes	No	100	5+	Central and eastern US, UK, Australia, Hong Kong	Niche player
HP	4	8	8	OpenStack	87	Good	Yes	No	99.95	2	Eastern and western US	Niche player
IBM	4	9	7	OpenStack	182	Good	Yes	No	100	5+	US, Netherlands, Singapore	Niche player
Google	4	17	13	Proprietary	42	Adequate	Yes	No	99.95	2	Central US, Belgium, Taiwan	Visionary
Others†	24	45	31	—	—	—	—	—	—	—	—	—

** In these columns, the sum of the percentages is greater than 100 because some companies use several products.*
† This group includes PaaS (platform as a service) and recent providers.

mainly, lack of isolation and lost robustness. In any multitenant environment, an instance's performance can be affected by your neighbors. A usage burst in a neighbor's instance can affect the available resources, notably compute units and disks' IOPS (I/O operations per second). Your architecture should deal with these changes.

Also, bottlenecks might arise owing to latency issues, even within instances at the same datacenter. Cloud providers offer some features to deal with this (for example, AWS placement groups). However, if your architecture has servers at different regional datacenters, you should use other techniques (for example, caching).

Security

Because of a public cloud's open characteristics, designing and maintaining a secure infrastructure should be an important driver in any cloud deployment. Enforce well-established security practices: firewalls, minimal server services to reduce attack vectors, up-to-date operating systems, key-based authentication, and so on. But challenges might arise from the increased number of servers to maintain and the use of the cloud for different development environments: development, staging, and

production. In this scenario, isolating and securing each environment is important because a breach in a prototyping server can give access through the secret keys to the whole infrastructure.

Monitoring

The ease of deploying new resources can make the number of servers grow exponentially. This raises new issues, and monitoring tools are vital to system management. First, they play a basic role in automatic scaling on a cyclical basis and based on events. Second, they're part of the tools needed to ensure a robust architecture, as the Netflix Chaos Monkey showed. Finally, they're important for detecting security breaches and forensic investigation, as some security breaches have shown.

Public Clouds

The public cloud was the first type of cloud offered to the general public, when AWS offered its experience with its private cloud to the general public. When you're selecting a vendor, it's important to consider several factors, mainly cost, performance, features, data location, and availability. But because the public cloud is a fairly recent technology, you should also consider vendor positioning and future use trends (see Table 1).

Cloud providers are battling for market position, which is leading them to reduce their public IaaS cloud prices, offering attractive solutions.

It's important to select the most effective vendor from a performance–cost perspective. However, your comparison should also consider whether the performance level is guaranteed, startup times, scalability responsiveness, and latency. These factors might vary among providers and impact the infrastructure's responsiveness.

The datacenters' location can affect your decision. The provider should comply with data privacy laws and corporate policies; the server locations should be based on these considerations. These restrictions might vary among countries and among companies. You might find you'll need to have all data under the same jurisdiction (for example, in Europe). In other cases, Safe Harbor principles, in which US companies comply with EU laws, can be good enough.

Understanding each player's SLA is important. But because almost every provider offers high-enough service levels (more than 99.95 percent), it's important to evaluate the accountability the SLA offers in case of noncompliance. Normally, this won't cover the costs of outages, so your infrastructure should be prepared for them.

Providers

Once you've defined your selection criteria, you can compare providers. The following are the most relevant ones.

Amazon. AWS (http://aws.amazon.com) continues being the dominant player in cloud computing, thanks to Amazon having been the first company to offer cloud services, in 2006.

AWS is cost effective. Its pay-as-you-go model lets you scale cloud capacity up or down without paying a high price. It also offers many additional IaaS services and integrated monitoring tools. It's particularly valuable for startups and agile projects requiring quick, cheap processing and storage.

Because AWS is a general provider, you can operate independently, which is convenient for normal operations but becomes risky when problems occur. Extensive technical support is a premium feature, whereas most of AWS's competitors offer it as a standard feature.

Microsoft Azure. Azure (http://azure.microsoft.com/en-us) entered the cloud IaaS market in February 2010. It has a large market share and is a good candidate because of its market position in other areas. It offers compute and storage services similar to those of other IaaS providers, and it allows full control and management of virtual machines. Additionally, Azure's UI is easy to use, especially for Windows administrators. However, because the Azure offering is newer than Amazon's or Rackspace's, it still has many features in "preview" mode and still has networking and security gaps.

Rackspace. Rackspace (www.rackspace.com) is a founder of OpenStack (which we describe later) and a major player in open source cloud IaaS. It hosts more than half of the Fortune 500 companies at its datacenters, while strongly focusing on SMEs (small-to-medium enterprises).

Rackspace provides an inexpensive, intuitive cloud with optional managed services and an easy-to-use control panel that suit SMEs. It also guarantees extensive support. However, it has limited pricing options, providing only month-to-month subscriptions. Also, it doesn't offer specialized services.

Google. Although Google AppEngine was a pioneer of cloud computing in the PaaS model, Google Compute Engine (https://cloud.google.com/compute) is relatively new

TABLE 2

The main products used to create private clouds.*

Solution	Reported usage (%)[4]	License	Trend, 2013–2014[4]
VMware	43	Proprietary	Down
OpenStack	12	Open source	Up
Microsoft	11	Proprietary	Down
CloudStack	6	Open source	Down
Eucalyptus	3	Open source	Down

** Some of the included solutions, as stated in* State of the Cloud Report,[4] *don't strictly meet all cloud-computing requirements.*

to the IaaS market. Nevertheless, Google's number of physical servers and global infrastructure make it a good candidate. Moreover, Google Compute Engine is well integrated with other Google services such as Google Cloud SQL and Google Cloud Storage.

Google Compute Engine is well suited for big data, data warehousing, high-performance computing, and other analytics-focused applications. Its main limitation is that it doesn't integrate administrative features. So, users must download extra packages.

HP. HP is still relatively new in the IaaS game; it launched its service in December 2012. Its public cloud, HP Cloud Compute (www.hpcloud.com), is built on OpenStack and offers a range of cloud-related products and services. It's a good candidate owing to its positioning in the server market. Its IaaS offering supports public, hybrid, and private clouds.

HP Cloud Compute is a good solution for companies that want to integrate their existing IT infrastructure with public-cloud services and invest in a hybrid cloud.

IBM. IBM's resources, size, and knowledge of datacenters make it another player to consider. IBM Cloud (www.ibm.com/cloud-computing/us/en) offers core computing and storage services. This IaaS is best for large enterprises with heavy data-processing needs and security concerns.

IBM Cloud provides a good combination of management, software, and security features for administrators. However, its focus is limited to medium-to-large enterprises and enterprises whose main provider is IBM.

Issues and Concerns

When considering adoption of a cloud architecture, it's important to understand what the technology can offer you and the main issues you'll have to deal with in each of these new infrastructures. Only by clearly understanding each of the approaches' business and technical opportunities and limitations will you be able to select the best option on the basis of your needs.

Besides the economic advantages from a cost perspective, the main competitive advantages are the flexibility and speed the cloud architecture can add to your IT environment. In particular, this kind of architecture can provide faster deployment of and access to IT resources, and fine-grain scalability.

A recent survey indicated the issues that beginner and experienced enterprise cloud users face.[4] For beginners, the main issues are security, managing multiple clouds, integration with current systems, governance, and lack of expertise. Experienced companies face issues of compliance, cost, performance, managing multiple clouds, and security.

The differences are understandable. Different problems arise on the basis of the degree of advancement of cloud architecture adoption. Early on, the main issues are resource expertise and control because the company hasn't acquired enough knowledge of and experience with the architecture. For more experienced companies, performance and cost are important because the architecture's limitations might have started emerging.

Both groups must deal with security, compliance, and managing multiple clouds. Regarding security and compliance, some problems might arise from the multitenant architecture. Some of these problems might not be solved, which might tip the balance toward a private or hybrid cloud. Such a decision is plausible, in keeping with the issue of managing multiple clouds.

Private and Hybrid Clouds

To solve the issues with public clouds, cloud-computing providers introduced the private cloud. This cloud might be in the organization's buildings, in the farm of the organization's provider, or in another provider's datacenter. Usually, it will be virtualized, but other combinations are possible. The important element is that only the customer's organization can operate it.

Because all private-cloud products allow integration with public clouds, we discuss both private and hybrid clouds here. Table 2 shows the main products used to create private clouds.

Eucalyptus

Eucalyptus (www.eucalyptus.com) released its first product in 2008. Nowadays the company provides its software as open source products and services. (Recently, Eucalyptus was bought by HP, a supporter of OpenStack.) From the company's download area, you can install a private cloud on your computer. From its product area, you can contract servers for your private cloud.

Eucalyptus software's main advantage is its AWS compatibility (see Table 3), based on a partnership with Amazon. So, some features that AWS makes available for the public cloud are applicable to Eucalyptus services.

Eucalyptus software's weak points are the limited GUI and the risk of uncertainty generated by AWS's private-cloud strategy: AWS offers Amazon Virtual Private Cloud and a connection to a hardware VPN (virtual private network).

TABLE 3

Eucalyptus compatibility with Amazon Web Services (AWS).

AWS services	Eucalyptus components
Amazon Elastic Compute Cloud (EC2)	Cloud Controller
Amazon Elastic Block Storage (EBS)	Storage Controller
Amazon Machine Image (AMI)	Eucalyptus Machine Image
Amazon Simple Storage Service (S3)	Walrus Storage
Amazon Identity and Access Management (IAM)	IAM
Auto Scaling	Auto Scaling
Elastic Load Balancing (ELB)	ELB
Amazon CloudWatch	CloudWatch

OpenStack

OpenStack (www.openstack.org) is the other main player in the private-cloud field. It's also open source, and its greatest strength is its support from companies such as AT&T, AMD, Cisco, Dell, HP, IBM, Intel, NEC, Red Hat, VMware, and Yahoo.

OpenStack is complex, with different components and multiple command-line interfaces. Competitors say it's not a product but a technology. This can be a barrier for nontechnical companies but not for public- and private-cloud providers, which are OpenStack's main users. For them, an open source product is attractive because, just as with using Linux in server computers, there are cost and portability advantages for the end user.

Portability is another important feature of OpenStack because end users don't want to be locked into a particular provider. However, providing the option of portability can be an issue for providers that want to offer differentiated proprietary features.

CloudStack

Citrix purchased CloudStack (http://cloudstack.apache.org) from Cloud.com. Citrix donated it to the Apache Software Foundation, which released it after it spent time in the Apache Incubator.

Unlike OpenStack, CloudStack offers a complete GUI and a monolithic architecture that simplifies installing and managing the product. Like OpenStack, most installations belong to service providers. CloudStack also offers AWS compatibility through an API translator.

Proprietary Solutions

VMware (www.vmware.com) and Microsoft (www.microsoft.com/enterprise/microsoftcloud) emphasize the hybrid nature of their offerings. They have products for both public and private clouds and provide on-premises servers.

VMware products include vCloud Hybrid Service, vCloud Connector, and vSphere virtualization. Microsoft has Windows Azure, Windows Server, and Microsoft System Center.

These two providers offer a more integrated solution because they own their products, but the disadvantage is lack of portability.

The public-cloud market has some years of history and well-known players. But remember that the cloud-computing market is growing. Newcomers are always entering, and the leaders in public- and private-cloud services can change.

So, your selection of a cloud-computing model and provider must take into account the factors listed in Tables 1 and 2, a service's specific purpose, and the elements of the application you want to migrate to the cloud. The approach and reach of your cloud adoption efforts will be limited by each situation. For example, your application architecture and the technology involved won't be the same if you're migrating an application not yet developed or an existing legacy system.

Regarding a new application,

BUNTPLANET

BuntPlanet (http://buntplanet.com) is a small-to-medium enterprise focusing on software engineering. It was founded in 2000 in San Sebastian, Spain. It offers a range of applications mainly for utilities, and it develops custom applications using agile practices.

Although BuntPlanet had some experience with cloud services, it was only in 2009 that it decided to use the cloud for its internal applications. The availability of Amazon's European datacenter was a determining factor because BuntPlanet could comply with Spanish data protection laws. During the selection process, the company compared other alternatives but chose Amazon mainly for the extensive features and competitive price.

At first, BuntPlanet replicated its server architecture in the cloud with Amazon EC2 (Elastic Compute Cloud) servers. Since then, it has incrementally refactored its applications to realize the full potential of a cloud architecture. It now uses other Amazon services such as S3 (Simple Storage Service) for data storage, Glacier for backups, RDS (Relational Database Service) for relational databases, SNS (Simple Notification Service) for push notifications, and SQS (Simple Queue Service) for queues.

To achieve the required service level, BuntPlanet uses reserved instances and a second redundant region in Europe. It's also self-monitoring its cloud resources and has set alerts for failures. This approach lets the company achieve its desired reliability and availability.

BuntPlanet's experience with the public cloud has been positive from economical and technical viewpoints. Using a public cloud has allowed BuntPlanet to simplify processes and minimize the need to support a big hardware infrastructure and a high bandwidth in its installations. As a result, it's promoting this architecture in its customer projects, using the public cloud for load tests and production environments.

you should develop it with an elastic architecture and best practices in mind. Decouple the presentation, business, and logic layers in several services and use a queue system to communicate between them. A high number of servers, a fault-tolerant design, and automatic provisioning will require high-level features from the cloud provider or technology.

Regarding a complete legacy system, refactoring the application to achieve decoupling isn't feasible. A pure cloud architecture is impossible, and a reduced list of features is required. Your priority should be virtual instances' robustness and reliability.

Other scenarios, such as disaster recovery or using the cloud when the demand spikes (cloud bursting), require specific cloud technology features.

If you're dealing with a new application and provider independence is a priority, you might prefer an OpenStack provider. If you're migrating a legacy system and you have IT experience with VMware, you might select VMware for your cloud. Regarding cloud bursting in a Microsoft Server IT environment, you might choose the Microsoft solution. However, AWS, a market leader and proven feature-rich platform, is always an option.

As you can see, because of the variety of choices, different customers might choose different platforms. For example, HP and Rackspace (service providers), Cybercom (a consulting company), and eBay (an end user) use OpenStack, whereas VMware and Microsoft customers use their provider's solution. For a look at how one company (BuntPlanet) chose its cloud provider, see the sidebar.

References

1. M. Vouk, "Cloud Computing—Issues, Research and Implementations," *J. Computing and Information Technology*, vol. 16, no. 4, 2008, pp. 235–246.
2. P. Mell and T. Grance, *The NIST Definition of Cloud Computing*, US Nat'l Inst. of Standards and Technology, 2011.
3. F. Fehling, F. Leymann, and R. Retter, "Your Coffee Shop Uses Cloud Computing," *IEEE Internet Computing*, vol. 18, no. 5, 2014, pp. 52–59.
4. *State of the Cloud Report*, RightScale, 2014; www.rightscale.com/lp/2014-state-of-the-cloud-report.
5. T. Rodrigues, "Top Cloud IaaS Providers Compared," *Enterprise Cloud*, 27 Aug. 2013; www.techrepublic.com/blog/the-enterprise-cloud/top-cloud-iaas-providers-compared.
6. "Vendor Landscape: Cloud Infrastructure-as-a-Service," Info-Tech Research Group, 2014; www.infotech.com/research/ss/select-the-right-cloud-infrastructure-server-partner.
7. *Magic Quadrant for Cloud Infrastructure as a Service*, Gartner, May 2014; www.gartner.com/technology/reprints.do?id=1-1UKQQA6&ct=140528&st=sb.

NICOLÁS SERRANO is a professor of computer science and software engineering at the University of Navarra. Contact him at nserrano@tecnun.es.

GORKA GALLARDO is a professor of information systems at the University of Navarra. Contact him at ggallardo@tecnun.es.

JOSUNE HERNANTES is a professor of computer science and software engineering at the University of Navarra. Contact her at jhernantes@tecnun.es.

IMPACT

Editor: **Michiel van Genuchten**
VitalHealth Software
genuchten@ieee.org

Editor: **Les Hatton**
Kingston University
l.hatton@kingston.ac.uk

The Software behind Moore's Law

Rogier Wester and John Koster

ASML is the world's leading provider of lithography systems for the semiconductor industry. Its scanners now contain around 45 million lines of software, one of the largest software systems discussed in the Impact series so far. —*Michiel van Genuchten and Les Hatton*

IN ASML'S LATEST technology challenge, a larger than 20 kW CO_2 laser vaporizes tin droplets running at 50 kHz. The resulting EUV (extreme UV) light is collected and transported into a scanner that produces wafers from which microchips are made. A sophisticated mirror path guides the light via a reflective mask, which defines the final imprint in photoresist on the wafer. Factory automation cycles the wafers between various lithography processing steps, involving multiple scanner types and analysis tools. The outcome is a fully processed wafer containing the chips for the end-user applications. Figure 1 shows the machine that handles this process.

EUV technology is the next stepping stone to keep up with Moore's law: every 24 months, the number of transistors that can be placed on a chip doubles[1] (see Figure 2). However, to maintain Moore's law, chip shrinkage must continue through continuous technology upgrades. These innovations form the basis for new-product platforms, which improve performance in terms of wafer output (throughput), nanometer precision (overlay), and imaging capabilities (focus and critical dimension uniformity). At the same time, these upgrades typically address more and more system aspects and their dependencies.

However, hardware can't be perfected at the nanometer level. So, software control algorithms correct hardware imperfections on the basis of mathematical models that specify the (inverse) behavior. In this way, hardware imperfections can be overcome as long as they're reproducible. Consider the reticle providing the print to be exposed on the wafer. During exposures, laser light heats the reticle, deforming the print. To keep the print straight, ASML employs thermal models of the reticle.

Besides technical challenges, chip production must deal with manufacturing and the cost of ownership. Moore's law happens only if the cost per gate also shrinks proportionally. To achieve this, ASML scanners are fully automated, high-speed factories. The software controls the transition from customer job to exposed wafer.

0740-7459/15/$31.00 © 2015 IEEE

FIGURE 1. The lithography scanner on the left is attached to the track in the middle, which is attached to the metrology system on the right. Wafers are coated with photoresist in the track, measured and exposed in the lithography scanner, returned to the track for development, and analyzed by the metrology system. (Image © ASML; used with permission.)

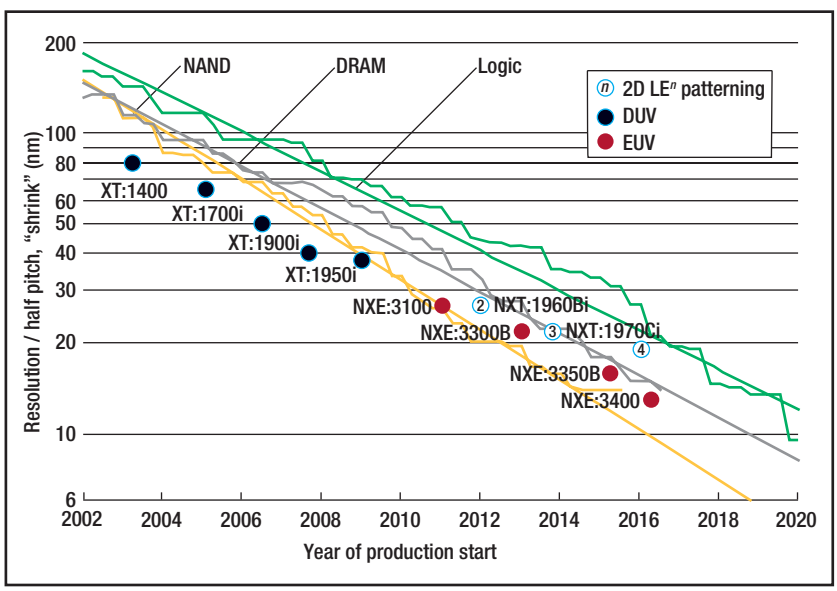


FIGURE 2. Chip shrinkage related to the development of ASML scanning technologies over time.[2] XT, NXT, and NXE are product families. DUV is deep UV light; EUV is extreme UV. LE means Litho-Etch; *n* is the number of exposure iterations per layer. (Figure based on image © ASML.)

Software for ASML Scanners

Software controls the scanner's execution; abstract state machines drive the operation steps. The software consists of 45 MLOC—about 2,000 components connected through 9,000 interfaces, and 60,000 files written in multiple programming languages. In the final system configuration, 20 computing nodes execute 200 processes. Because the software must support both old and new systems, its development has spanned more than 10 years. Older systems are frequently upgraded to improve performance and maintain compatibility with newer systems.

The scanner's software architecture (see Figure 3) shows all major functions of the machine:

- The *stages and handlers* module takes care of the wafer- and reticle-handling subsystems.
- The *optical path* module corresponds to the light control and transportation subsystems.
- The *sensors* module provides imaging-control information.
- The *environmental control* module deals with crosscutting environmental aspects.
- The *computing platform and facilities* module provides the computing infrastructure and software facilities.
- The *system functions* module covers top-level material routing, imaging control, and the interface to the MES and LIS.

Currently, three software product lines support the TWINSCAN product family: NXE employs EUV technology, and XT and NXT employ DUV (deep UV) technology. These product lines have about one-third of the code in common (see Figure A in the sidebar). Changes to this common code must be tested in all product contexts. The common code is primarily in the modules for the computing platform and facilities, sensors, and system functions.

The product lines also have distinct features. For example, the optical-path module in the DUV immersion variant supports a film of water between the lens and wafer. In the EUV variant, mirrors in a vacuum reflect the light. The software is organized and constructed such that technology-specific features can't influence one another. That is, specific EUV and DUV technology features don't coexist in the final software application that controls the machine.

Software Development

The scanner is decomposed into 45 multidisciplinary functions, of which 35 require software. Some of these 35 consist solely of software; the others also require electronics, mechanics, mechatronics, thermal conditioning, and so on. The ASML development organization is based on this functional decomposition. Dedicated function owners are integrally responsible for all parts (hardware and software) developed in the functions.

ASML employs modularity to keep the software's size and comprehensibility manageable. The structure between the functions is specified in an architecture description language. Changes to the structure require the responsible software architects' explicit approval. A build failure results when dependencies fall outside the imposed structure. This enforced control is necessary to maintain architectural integrity in an organization of 800 software engineers.

For new-product development, hardware and software development is optimized for smooth machine integration. So, software is always present at the system level, even during machine build-up. The software deals with absent hardware modules; to support this, all components can be put into a special simulation mode.

To avoid lost time on scarce hardware prototypes, software is first tested on other platforms. The simplest is the *devbench*: a single multicore computer representing the main control host, with other components (those that run on other hosts) running in simulation mode. The next level is the *testbench*, which includes the main control host and all electronic racks of the full machine. Finally, there are the full hardware prototypes, which can be in varying states of integration. Availability of the full machine for software testing is limited and sometimes nonexistent—for example, when the machine types are only at customer sites.

Products in the field require frequent software upgrades for adding functionality and repairing bugs. These upgrades are released in a series of service packs that provide fully qualified software increments. In emergencies, a single dedicated fix (or patch) is released immediately. These upgrades are also embedded in the mainstream development archive. So, new software releases not only drive new machine

SOFTWARE GROWTH OVER TIME

Figure A shows our software's growth over time. In early 2000, the software was rewritten from scratch. The first dual-stage scanners shipped in 2003, with a modest code size. The recent NXT flagship product line takes 34 MLOC, of which 18 MLOC are shared with other product lines. In 2013, the XT product line was placed in a separated archive, causing the dip. The current archive of 45 MLOC contains the code for both the NXT and NXE product lines. The increase from 5 to 45 MLOC indicates a compound annual growth rate of 1.17—the median of the growth rates found previously,[1] and comparable to medical scanners.[2]

References

1. M. van Genuchten and L. Hatton, "Quantifying Software's Impact," *Computer*, vol. 46, no. 10, 2013, pp. 66–72; doi:10.1109/MC.2013.7
2. L. Hofland and J. van der Linden, "Software in MRI Scanners," *IEEE Software*, vol. 27, no. 4, 2010, pp. 87–89; doi:10.1109/MS.2010.106.

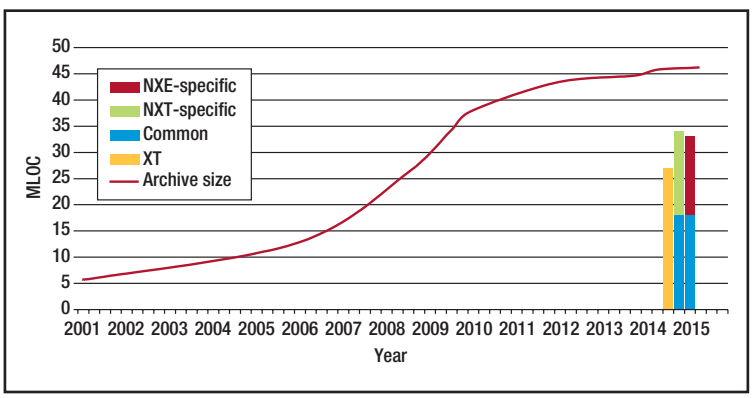


FIGURE A. The TWINSCAN software's size over time. The software must support both old and new systems; older systems are frequently upgraded to improve performance and maintain compatibility with newer systems. (Figure based on image © ASML.)

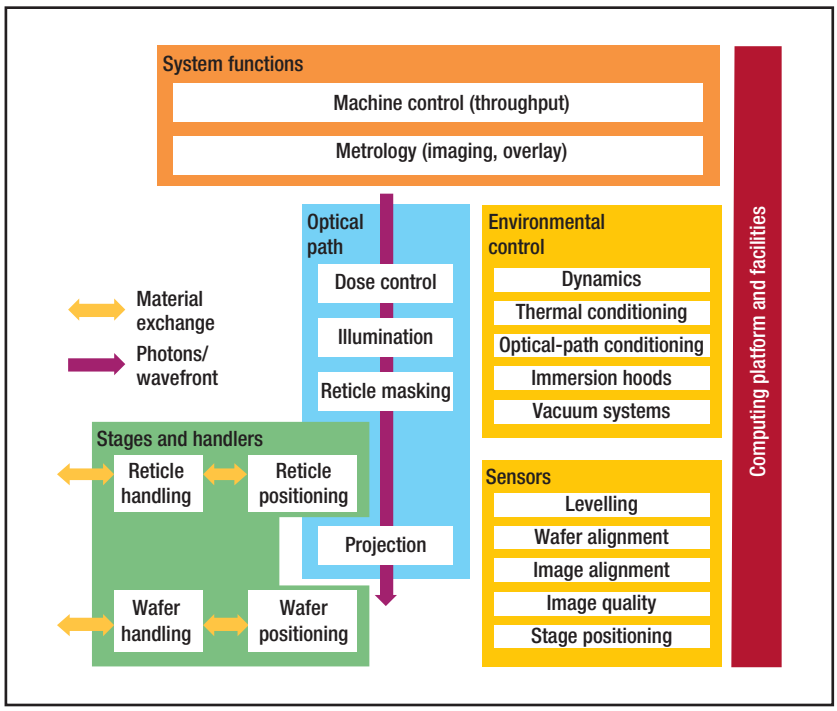


FIGURE 3. The high-level functions in the ASML scanner. The software architecture reflects the factory control and automation process. (Figure based on image © ASML.)

configurations but also replace older versions, thereby limiting the number of releases the development organization must actively support.

Development Challenges

This approach motivates reuse of software in new-product development. The use of variation points enables specific product configurations. At the same time, though, the developer's freedom is limited because old configurations shouldn't break. This leads to a situation in which software size and complexity gradually grow with new product configurations, making timely refactoring of legacy software more important but more difficult. In other words, a conflict exists between short-term innovation cycles and paying off longer-term technical debt.[3]

The introduction of new technologies typically proceeds through new-component development. For targeting existing components, an accurate specification of the component interface is required. If such a specification isn't available, the interface behavior must be reengineered, which is difficult. This involves introducing *armoring patterns* that monitor newly designed interfaces' behavior. Typically, for a new-product platform, one-third of the existing components are untouched. The other components are candidates for improvement.

The continuing application of Moore's law is inextricably linked with software, particularly as software infrastructure outlives scanner hardware. Software technology innovations are a fact of industry, and seamless integration with legacy software is required to combine speed of innovation with already mature products. Technology independence and cohabitation of different solutions are steps to support this, as are formal verification techniques that mathematically prove compliance of the component interface and implementation. At ASML, higher abstraction levels and executable specifications are the directions we're taking to master the future software growth necessary to implement Moore's law.

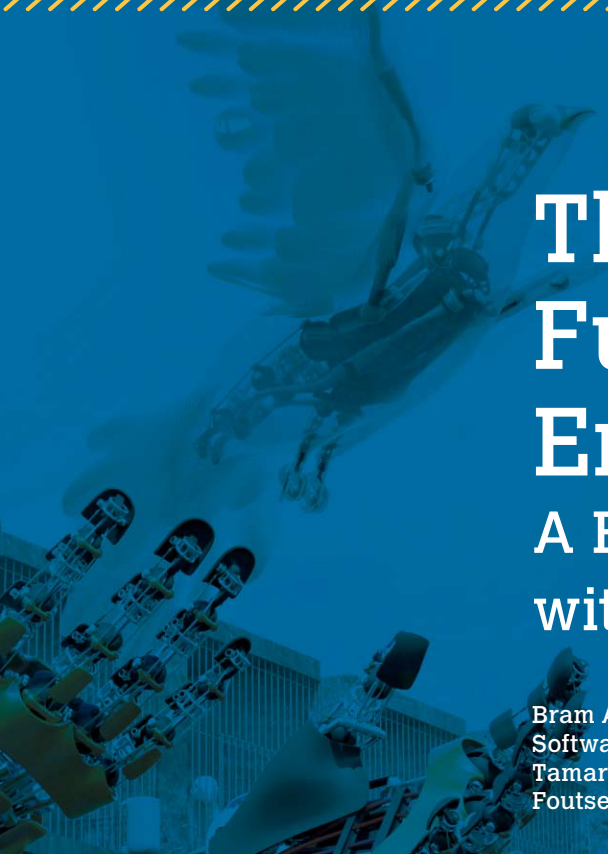
References

1. "Moore's Law Inspires Intel Innovation," Intel; www.intel.com/content/www/us/en/silicon-innovations/moores-law-technology.html.
2. S. Miller, "ASML's NXE Platform for Volume Production," presentation at Semicon West 2013, 2013; www.semiconwest.org/sites/semiconwest.org/files/docs/SW2013_Skip%20Miller_ASML.pdf.
3. P. Kruchten, R. Nord, and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *IEEE Software*, vol. 29, no. 6, 2012, pp. 18–21.

ROGIER WESTER is a senior manager of software development at ASML. Contact him at rogier.wester@asml.com.

JOHN KOSTER is a senior director of software development at ASML. Contact him at john.koster@asml.com.

FOCUS: GUEST EDITORS' INTRODUCTION



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

AUDIO

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.

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

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.

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.

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

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

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?

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

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.

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.

See www.computer.org/software-multimedia for multimedia content related to this article.

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.

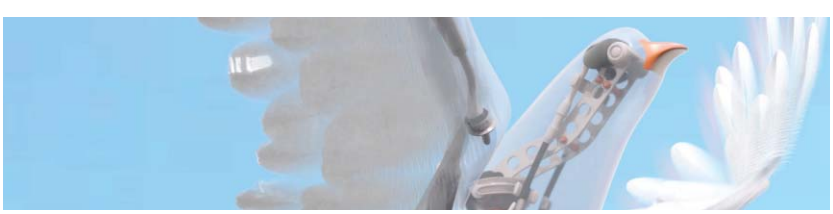
KIM MOIR is a release engineer at a Mozilla. Her research interests are build optimization, scaling large infrastructure, and optimizing build and release pipelines. Moir received a Bachelor of Business Administration from Acadia University. Contact her at kmoir@mozilla.com.

Continuous Delivery

Huge Benefits, but Challenges Too

Lianping Chen, Paddy Power

// This article explains why Paddy Power adopted continuous delivery (CD), describes the resulting CD capability, and reports the huge benefits and challenges involved. This information can help practitioners plan their adoption of CD and help researchers form their research agendas. //



CONTINUOUS DELIVERY (CD) is a software engineering approach in which teams keep producing valuable software in short cycles and ensure that the software can be reliably released at any time. CD is attracting increasing attention and recognition.

CD advocates claim that it lets organizations rapidly, efficiently, and reliably bring service improvements to market and eventually stay a step ahead of the competition.[1] This sounds great. However, implementing CD can be challenging—especially in the context of a large enterprise's existing development-and-release environment.

Here, I explain how we adopted CD at Paddy Power, a large bookmaking company. I describe the resulting CD capability and report the huge benefits and challenges involved. These experiences can provide fellow practitioners with insights for their adoption of CD, and the identified challenges can provide researchers with valuable input for developing their research agendas.

The Context

Paddy Power is a rapidly growing company, with a turnover of approximately €6 billion and 4,000 employees. It offers its services in regulated markets, through betting shops, phones, and the Internet.

The company relies heavily on an increasingly large number of custom software applications. These applications include websites, mobile apps, trading and pricing systems, live-betting-data distribution systems, and software used in the betting shops. We develop these applications using a range of technology stacks, including Java, Ruby, PHP, and .NET. To run these applications, the company has an IT infrastructure consisting of thousands of servers in different locations.

These applications are developed and maintained by the Technology Department, which employs about 400 people. A software development team's size depends on the application's size and complexity. Our teams range from two to 26 people; most teams have four to eight people.

The release cycle for each application also varies. Previously, each application typically had fewer than six releases a year. For each release cycle, we gathered the requirements at the cycle's beginning. Engineers worked on development for months. Extensive testing and bug fixing occurred toward the cycle's end. Then, the developers handed the software over to operations engineers for deploying to production. The deployment involved many manual activities.

This release model artificially delayed features completed early in the release cycle. The value these features could generate was lost, and early feedback on them wasn't available.

0740-7459/15/$31.00 © 2015 IEEE

Many releases were a "scary" experience because the release process wasn't often practiced and there were many error-prone manual activities. Priority 1 incidents caused by manual-configuration mistakes weren't uncommon. In addition, the release activities weren't efficient. Just setting up the testing environment could take up to three weeks.

To improve the situation, Paddy Power started an initiative to implement CD. The company established a dedicated team of eight people, which has been working on this for more than two years.

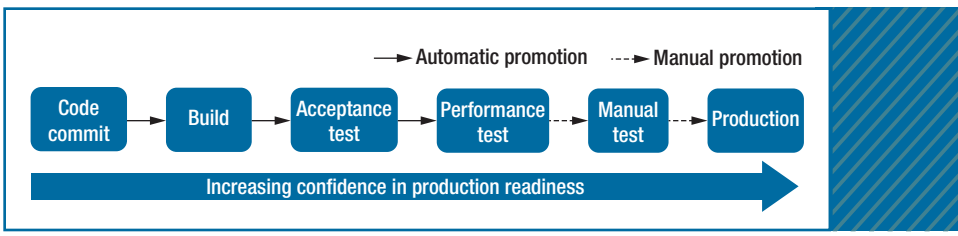


FIGURE 1. An example continuous delivery (CD) pipeline. Promotion—advancing the pipeline's execution from one stage to another—can be automatic or manual. Our confidence in a build's production readiness increases as the build passes through each pipeline stage.

The CD Pipeline

Because we needed to support many diverse applications, we built a platform that lets us create a CD pipeline for each application. Our team operates and maintains this platform. When an application development team needs a new CD pipeline for its application, we create one.

An application's pipeline might differ slightly from another application's pipeline, in terms of the number and type of stages, to best suit that application. Figure 1 shows an example pipeline.

Code Commit

The code commit stage provides immediate initial feedback to developers on the code they check in. When a developer checks in code to the software-configuration-management repository, this stage triggers automatically. It compiles the source code and executes unit tests.

When this stage encounters an error, the pipeline stops and notifies the developers. Developers fix the code, get the changes peer reviewed, and check in the code. This triggers the code commit stage again and starts a new execution of the pipeline. If everything goes well, the pipeline automatically moves to the next stage.

Build

The build stage executes the unit tests again to generate a code coverage report, runs integration tests and various static code analyses, and builds the artifacts for release. It uploads the artifacts to the repository that manages them for deployment or distribution. All later pipeline stages will run with this set of artifacts.

Before we moved to CD, the binaries released to production might differ from the tested binaries. This was because we built the software multiple times, each for a different stage. Each time we built the software, we ran the risk of introducing differences. We've seen the bugs these differences cause. Fixing them was frustrating because the software worked for the developers and testers but didn't work in production. The CD pipeline eliminated these bugs.

If anything goes wrong, the pipeline stops and notifies the developers; otherwise, it automatically moves to the next stage.

Acceptance Test

The acceptance test stage mainly ensures that the software meets all specified user requirements. The pipeline creates the acceptance test environment, a production-like environment with the software deployed to it. This involves provisioning the servers, configuring them (for example, for network and security), deploying the software to them, and configuring the software. The pipeline then runs the acceptance test suite in this environment.

Previously, setting up this environment was a manual activity. For one of the very complex applications, setup took two weeks of a developer's time. Even for a smaller application, it took up to half a day.

For a new project, setup took even longer. The developers needed to request new machines from the infrastructure team, request that the Unix or Windows engineering team configure the machines, request network engineers to open connections between the machines, and so on. This could take a month.

With the CD pipeline, the developers don't need to perform these activities. The pipeline automatically sets up the environment in a few minutes.

Similar to the other stages, if any errors arise, the pipeline stops and notifies the developers; otherwise, it moves to the next stage.

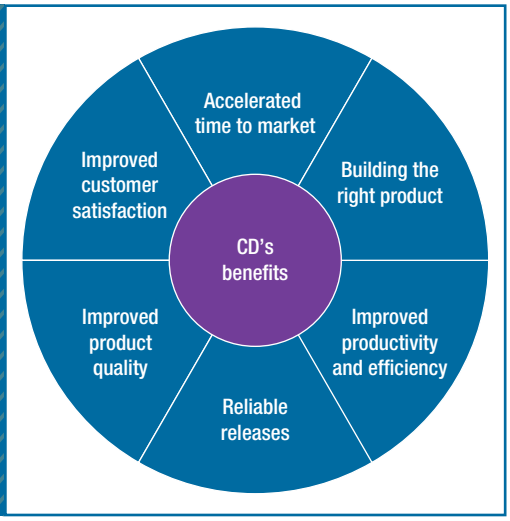


FIGURE 2. CD's benefits. Motivated by these benefits, the company is increasing its investment in CD.

Performance Test

The performance test stage gauges how the code change will affect the software's performance. The pipeline sets up the performance test environment, runs a suite of performance tests in this environment, and feeds the results into the tool that centrally manages software quality.

Previously, owing to the considerable effort of setting up a performance test environment, performance testing didn't occur during development. We performed it only before the big release.

With the pipeline, performance testing occurs for each code commit that passed the previous stages. This lets developers get immediate feedback if the code change has degraded the software's performance. Diagnosing and fixing problems at this time is much cheaper than doing so before a big release.

Manual Test

Although our automated testing is quite comprehensive, manual testing is sometimes necessary (for example, when the testers perform exploratory testing[2] and the business users perform user acceptance testing).

Previously, the testers had to set up a manual-testing environment, which they said was a headache. There were many manual, error-prone steps.

With CD, they no longer need to worry about this. The pipeline automatically sets up the test environment and notifies the testers with email containing the information required to access the deployed application.

When the tests complete satisfactorily, the set of artifacts is promoted from "potential release candidate" to "release candidate." At this point, the software has passed all the quality checks and is ready to deploy to production.

Production

Deployment into production takes just the click of a button.

Previously, such deployment sometimes failed because of errors in the deployment process and scripts. CD has no manual deployment steps, and the deployment process and scripts have been tested many times in previous stages.

Benefits

So far, we've moved 20 applications to CD. They're developed by one of the largest software development groups. Their main users are business people in the company. All the development teams have adopted an agile approach called Kanban[3] while moving their applications to CD.

On these applications, CD has produced the following six main benefits (see Figure 2).

Accelerated Time to Market

The release frequency has increased dramatically. Previously, an application released once every one to six months. Now, an application releases once a week on average. Some applications have released multiple times a day when necessary.

The cycle time from a user story's conception to production has decreased from several months to two to five days.

CD lets us deliver the business value inherent in new software releases to our customers more quickly. This capability helps the company stay a step ahead of the competition, in today's competitive economic environment.

Building the Right Product

Frequent releases let the application development teams obtain user feedback more quickly. This lets them work on only the useful features. If they find that a feature isn't useful, they spend no further effort on it. This helps them build the right product.

Previously, teams might have worked on features that weren't useful but didn't discover that until after the next big release. By that time, they had already spent months of effort on those features.

Improved Productivity and Efficiency

Productivity and efficiency have also improved significantly. For example, developers used to spend 20 percent of their time setting up and fixing their test environments. Now, the CD pipeline automatically sets up the environments. Similarly, testers used to spend considerable effort setting up their test environments. Now, they don't need to do this, either.

Operations engineers used to take a few days' effort to release an application to production. Now, they only need to click a button; the pipeline automatically releases the application to production.

Furthermore, developers and operations engineers used to spend much effort on troubleshooting and fixing issues caused by the old release practice. The CD pipeline eliminated these issues. The effort that otherwise would have been spent fixing these issues can be used for more valuable activities.

Reliable Releases

The risks associated with a release have significantly decreased, and the release process has become more reliable.

As we mentioned before, with CD, the deployment process and scripts are tested repeatedly before deployment to production. So, most errors in the deployment process and scripts have already been discovered.

With more frequent releases, the number of code changes in each release decreases. This makes finding and fixing any problems that do occur easier, reducing the time in which they have an impact.

Moreover, the CD pipeline can automatically roll back a release if it fails. This further reduces the risk of a release failure.

The engineers commented that they don't feel the same level of stress on the release day that they did previously. That day becomes just another normal day.

Improved Product Quality

Product quality has improved significantly. The number of open bugs for the applications has decreased by more than 90 percent.

With CD, immediately after a code commit, the whole code base undergoes a series of tests. If the tests find a problem, the developers fix it before moving to another task. This eliminates many bugs that otherwise would have been open in the bug-tracking system with the old release practice.

Previously, the bug-tracking system recorded many open bugs. Approximately 30 percent of the workforce was fixing bugs. Now, usually nobody is working on customer-found bugs. Bugs are so rare that the teams no longer need a bug-tracking system.

> Frequent releases let the application development teams obtain user feedback more quickly.

On the rare occasion that a bug is discovered in production, it's added to the team's Kanban board and gets fixed and released in a few days. Before, customers had to wait for the next big release to get the bug fix. The time frame was usually months.

In addition, priority 1 incidents in production have decreased significantly. This is because, apart from the reasons we just listed, the CD pipeline has eliminated the errors that might result from manual configurations and error-prone practices.

Improved Customer Satisfaction

Before we moved to CD, distrust and tension existed between the users' department and the software development teams, owing to quality and release issues. The managers commented that the relationship has improved enormously. Trust has been established.

Challenges

Motivated by these huge benefits, the company is increasing its investment in CD. Expanding the adoption of CD across the company and improving the CD platform are receiving top priority. Nevertheless, implementing CD involves considerable challenges.

Organizational Challenges

The biggest challenge has been organizational. Release activities involve many divisions of the company. Each has its own interests, ways of working, and perceived territories of control. Tension existed between divisions due to competing goals. For example, we needed root access to the servers, and another team controlled this permission. Arriving at a solution involved much consultation and negotiation over six months.

To address the organizational challenges, the leadership team restructured the organization to break down barriers among teams and promote a collaborative culture. The situation has improved since.

Although literature on organizational change exists,[4] little, if any, research focuses on introducing CD to an organization. Further research on this topic—for example, understanding the challenges in more depth and developing strategies and practices to tackle them more effectively—will significantly help an organization's smooth adoption of CD.

RELATED WORK IN CONTINUOUS DELIVERY

Gerry Claps and his colleagues studied the technical and social challenges of adopting continuous delivery (CD).[1] Helena Olsson and her colleagues explored the barriers to transitioning from agile development to CD.[2] However, none of them covered the challenges of CD tooling development. I describe these challenges in the main article.

Mika Mäntylä and his colleagues performed a semisystematic literature review of rapid release (including CD).[3] They concluded that evidence of the claimed advantages of rapid release is scarce. In the main article, I provide what I believe is the first comprehensive evidence-based description of CD's benefits in the research literature.

References

1. G.G. Claps, R. Berntsson Svensson, and A. Aurum, "On the Journey to Continuous Deployment: Technical and Social Challenges along the Way," *Information and Software Technology*, vol. 57, 2015, pp. 21–31.
2. 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. 38th EUROMICRO Conf. Software Eng. and Advanced Applications* (SEAA 12), 2012, pp. 392–399.
3. M. Mäntylä et al., "On Rapid Releases and Software Testing: A Case Study and a Semi-systematic Literature Review," *Empirical Software Eng.*, Oct. 2014, pp. 1–42

ABOUT THE AUTHOR

LIANPING CHEN is a senior software engineer at Paddy Power and a part-time doctoral researcher at Lero—The Irish Software Engineering Research Centre at the University of Limerick. His research interests include software requirements and architecture, continuous delivery, DevOps, and software product lines. Chen received an MS in software engineering from Northwestern Polytechnical University. Contact him at lchen@paddypower.com.

Process Challenges

Many traditional processes hinder CD. For example, a feature that's ready for release normally must go through a change advisory board.[5] This can delay the release for up to four days. If a feature takes only a few days from conception to being ready for release, this four-day period accounts for too much of the feature's total cycle time.

Research is needed to identify these processes (covering areas of business, software development, operations, and so on) and develop and verify alternatives that suit CD.

Technical Challenges

A robust, out-of-the-box, comprehensive, and yet highly customizable solution for CD doesn't exist yet. So, we developed our own solution, which was costly. Tools that fill this gap will save companies considerable resources.

When we're building the CD platform, we use many different tools and technologies as building blocks. Avoiding vendor lock-in is challenging. Work on developing widely accepted standards, defining open APIs, and building an active plug-in ecosystem will help alleviate the challenge.

Dealing with applications that aren't amenable to CD (for example, large, monolithic applications) is also challenging. A huge number of such applications exist in the industry. Research is needed on understanding their characteristics and identifying and developing the best strategies or practices to tackle them.

We'd like to solve the challenges we just described through close collaboration with researchers and companies, so that more organizations can easily avail themselves of CD's benefits.

For a brief look at other research related to CD, see the sidebar.

Acknowledgments

I thank my colleagues, Klaas-Jan Stol, this article's reviewers, and the editors for their help and thoughtful comments. The article represents only my own views and doesn't necessarily reflect those of my employer.

References

1. J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley Professional, 2010.
2. C. Kaner, J. Falk, and H.Q. Nguyen, *Testing Computer Software*, 2nd ed., John Wiley & Sons, 1999.
3. D.J. Anderson, *Kanban: Successful Evolutionary Change for Your Technology Business*, Blue Hole Press, 2010.
4. R. Todnem By, "Organisational Change Management: A Critical Review," *J. Change Management*, vol. 5, no. 4, 2005, pp. 369–380.
5. A. Rob, *Effective IT Service Management: To ITIL and Beyond!*, Springer, 2007.

Why and How Should Open Source Projects Adopt Time-Based Releases?

Martin Michlmayr, Hewlett-Packard

Brian Fitzgerald and Klaas-Jan Stol, Lero—The Irish Software Engineering Research Centre

// Traditional release strategies have associated problems that can be overcome by time-based release management. Findings from interviews with key members of seven prominent volunteer-based open source projects reveal several benefits of adopting a time-based release strategy. //

FREQUENTLY RELEASING software helps improve its quality. Open source evangelist Eric Raymond phrased this succinctly as "Release early. Release often."[1] However, the software engineering community has devoted little attention to release engineering.[2] Although this is changing,[3] research on the topic tends to focus on proprietary software development;[4,5] few studies have focused on open source software.[6] (For more details, see the sidebar.)

Studying open source projects is useful for several reasons. The software industry depends increasingly on open source and is investing in and sponsoring successful open source projects.[7–9] So, an understanding of how such volunteer communities work is important because companies must be able to interact with them.[8] Also, companies can draw lessons from open source communities by studying best practices and applying them internally.[10–12]

Regarding releases, volunteer-driven open source projects usually employ one of two strategies. Many projects issue a new release after implementing a certain set of features,[13] which involves numerous challenges. Alternatively, projects might adopt a time-based strategy, in which releases are planned for a specific date.[14] Previous research has suggested that community activity increases close to release dates[15] and that new releases result in spikes of downloads.[16] Such a regular "heartbeat" signals to both users and potential contributors that the project is progressing in a healthy way. For instance, the date of the latest release may impact a potential contributor's first impression of the project;[17] attracting new developers is important for a project's sustainability. It also allows vendors and distributors to rely on a consistent stream of new releases and to include the latest features and bug fixes.

Here, we report on an interview study of release management that included seven prominent open source projects that have moved from feature-based to time-based releases. On the basis of our findings, we discuss why projects should adopt time-based releases and how they can adopt such a strategy.

0740-7459/15/$31.00 © 2015 IEEE

RELATED WORK IN OPEN SOURCE RELEASE MANAGEMENT

Few research studies have focused on release management in open source development. In one of the first studies, Justin Erenkrantz presented a taxonomy of open source release management that helps in understanding and comparing release strategies and applied the taxonomy to compare three projects: the Linux kernel, Subversion, and the Apache HTTP server.[1] (Hyrum Wright observed that all projects have changed their release process since then.[2])

Hyrum Wright and Dewayne Perry reported a qualitative case study on release mismanagement in the Subversion project.[3] They described how the new version at the time (ver. 1.5) was delayed owing to numerous setbacks, such as high complexity and many bugs. Later, Wright identified several challenges related to release engineering, pertaining to software architecture, release failure's social causes, and challenges related to the software domain and tools.[2,4]

Foutse Khomh and his colleagues investigated whether faster releases improve software quality.[5] Their study of Firefox focused on the relationship between release cycle length and postrelease defects, how quickly bugs were fixed, and the adoption rate by users. Our study, in contrast, focuses on why and how to adopt a time-based release strategy (see the main article).

References

1. J.R. Erenkrantz, "Release Management within Open Source Projects," *Proc. 3rd Workshop Open Source Software Eng.*, 2003, pp. 51–55.
2. H.K. Wright, "Release Engineering Processes, Their Faults and Failures," PhD dissertation, Univ. of Texas at Austin, 2012.
3. H.K. Wright and D.E. Perry, "Subversion 1.5: A Case Study in Open Source Release Mismanagement," *Proc. 2nd FLOSS Workshop*, 2009, pp. 13–18.
4. H.K. Wright and D.E. Perry, "Release Engineering Practices and Pitfalls," *Proc. 2012 Int'l Conf. Software Eng.* (ICSE 12), 2012, pp. 1281–1284.
5. F. Khomh et al., "Understanding the Impact of Rapid Releases on Software Quality: The Case of Firefox," *Empirical Software Eng.*, May 2014, doi:10.1007/s10664-014-9308-x.

The Study

Our study focused on large, distributed open source projects controlled by volunteers. Although some contributors may receive payments to work on a project, they cannot be controlled by a company. Therefore, we didn't include projects controlled by companies, such as the Ubuntu Linux distribution produced by Canonical. We interviewed 20 key informants, who all were knowledgeable about the respective release process through their roles as a core developer, release manager, quality assurance (QA) team member, project foundation board member, or steering committee member. We analyzed the transcribed interviews for patterns relating to release management, time frames, deadlines, and delays. Although all the studied projects intended to follow a time-based release schedule, some projects (for example, the Linux kernel) achieved this better than others (for example, Debian).

Table 1 describes these projects and the interviewees. The online supplement to this article (at http://doi.ieeecomputersociety.org/10.1109/MS.2015.34) provides additional information on the projects and interview guide.

Why Adopt Time-Based Releases?

Traditional release strategies deliver a new version of the software based on a set of new features or defect fixes. In theory, such feature-based releases can work well because developers have identified the work to do before the next release. Such feature-based releases could follow a short cycle such as sprint-driven development—Scrum, for example, works this way.

In practice, however, such feature-based strategies can cause a variety of interlinked problems, especially in volunteer-run open source projects. This can result in a long, unpredictable release cycle because certain features may never be finished, and consequently the release may not happen. Thus, feature-based releases are associated with a number of problems.

Unpredictable Release Schedules Cause Rushed Code

When using a feature-based strategy, an open source project might make a release even if not all planned features have been implemented. This can

TABLE 1

Open source projects investigated for this study.

Project	Type of software	When was the time-based strategy introduced?	Intended interval	Approximate size, mid-2014 (LOC)	Interviewees
Debian	Linux distribution	After v3.1, mid-2005	2 yrs.	324 M	Two release managers
GNU Compiler Collection (GCC)	Compiler	2001	6 mo.	6.5 M	Steering-committee member Distributor
GNOME	Desktop environment	Early 2003	6 mo.	8.2 M	Release manager for Java-GNOME Core developer Two release managers
Linux kernel	OS kernel	Mid-2005	2–3 mo.	17 M	Two maintainers Core developer
OpenOffice.org	Office productivity suite	Early 2005	6 mo.	9 M	Quality assurance team member Community manager Distributor
Plone	Content management system	Early 2006	6 mo.	550 K	Plone release manager Archetypes release manager Release manager
X.org	GUI window manager	Late 2005	6 mo.	2.3 M	Release manager Member of X.org Foundation board of directors Distributor / release manager

happen because work on many open source projects is voluntary, so nobody may have worked on these features. One core developer explained,

> *In an open source environment, the feature-based strategy is just basically impossible unless you want to wait forever …, which is what happens to a lot of projects. Some haven't had a release in five years. You cannot tell anybody to do anything.*

So, a release might be announced suddenly and unexpectedly. This results in "rushing in" code—what one Linux kernel developer called a "thundering herd of patches." This is because developers want to integrate their features and bug fixes in the latest release and have no idea when the next release will occur.

Having time as the criterion (rather than features that may or may not be completed) enables a project to plan and establish a predictable schedule. One release manager stated that, by making the schedule public, the project sends a message that

> *certain things will happen at certain times [and] you take away some of the mystery.*

A regular release also helps developers refrain from rushing code contributions. Developers wish to see their contributions included in a stable release as soon as possible as this improves their reputation and provides the satisfaction of seeing their code used by others. In a time-based strategy, they don't need to rush in code as the timing of the next release is known in advance. One participant explained,

> *For developers, regular releases are like trains: if you miss one, you know that there will be another one in the not-too-distant future.*

No Release Plan, No Adoption Plan

An unpredictable release schedule makes it difficult for adopters to plan. This is a particular concern for those users or corporations that act as consultants or component

integrators because they're unable to decide whether to use the latest stable (but possibly old) version or a more recent (but possibly less stable) development version. Vendors shipping an open source project as part of a distribution or larger product need to know when a release will be stable to ensure stability of the whole distribution or product.

Due to the uncertainty surrounding releases, many vendors avoid official development versions and work on their own versions. This leads to fragmentation between vendors (such as Linux distributions), which diverts significant resources from the official development releases. Not only is this inefficient duplication of work, it also has a substantial negative impact on these "sponsored" projects that benefit from company contributions.

A time-based strategy offers a predictable release schedule, which is of great benefit to vendors who distribute the software, as one participant explained:

> *You know the schedule in advance, and you can decide which version to ship. That's a really good point from the distributor's point of view.*

Infrequent Releases, Workload Accumulation

An infrequent, unpredictable release cycle causes developers to add features as they see fit, resulting in the accumulation of many changes over a significant period of time. Some projects, such as Debian and the Linux kernel, had long development phases. Many projects experienced additional delays resulting in numerous changes to be tested at the end of the development phase before a release. Consequently, getting the development tree stable in preparation for a release was difficult. As one QA team member explained,

> *It's a problem if you start packing too many features into a release. The features are there, but they are simply not stable.*

As the development version increasingly diverges from the latest stable release, fewer users are willing to test these releases. Most users don't start testing until a stable release is imminent, which would require an explicit plan or announcement. A long interval between stable releases therefore means that there's little feedback coming from the user community. Owing to the long time spent on development, some projects rushed out releases without sufficient testing or making test releases available for the development versions. This had a negative impact on the quality of the released software.

A time-based strategy ensures a more regular release cycle and creates a much tighter feedback loop with users. A core developer of GNOME commented,

> *If you can keep the development more tightly linked to the actual usage, you will get much better feedback in terms of bugs and development direction.*

Furthermore, more regular releases improve project members' proficiency in managing releases, which can help optimize the release process. Developers will become more used to the process, and release managers will become familiar with preparing a release, thus making the process more straightforward and standardized. One release manager illustrated this as follows:

> *With something like cooking, we're just used to doing it every day so you know how you do it and it doesn't fail.*

More regular releases improve project members' proficiency in managing releases.

Outdated Software, Outdated Bug Reports

Before the projects adopted time-based releases, stable releases used by end users were often quite old and outdated compared to the latest development version. Some projects, such as the Linux kernel, spend substantial effort to update their latest stable releases, particularly for urgent features such as new hardware support that would have to be back-ported from a development release to a stable release. While this provides benefits for end users, it limits the resources that could otherwise have been used to prepare a new stable release.

As a project's latest stable release becomes outdated, many bug reports often become of less value as they may already have been fixed in the latest development version. Even if a bug was highly critical, which would warrant a back-port of the fix to the stable release, developers

are unlikely to be motivated to invest time in doing so, given that their work is voluntary.

A time-based strategy implies a more regular stream of new versions and thus a more regular synchronization, not only among developers but also across the user base. This is because users become more used to updates to the latest release. For developers, both those within the project and others who integrate the software, it's important to synchronize regularly so that they work from the same code base to prevent merge conflicts. This also helps in keeping the issue database relevant as fewer bugs relating to older versions will be reported.

Delays Leading to Further Delays

Delays may manifest themselves in a vicious cycle leading to further delays. Once it becomes clear to developers that a release target can't be met or that deadlines aren't enforced, developers may try to push in more changes, thus destabilizing the code base and causing further delays. As intervals between releases increase, developers will realize that their new contributions won't be made available for a long time, causing them to push even harder to get the new code in.

Furthermore, these delays could lead to developer and end-user frustration. Developers who spend significant time and effort to complete their contributions could be left disillusioned if expected releases are constantly delayed, leaving their work unavailable to the general public. Given that many volunteers are driven by such things as fun and a sense of satisfaction, they may choose to spend their time on a project in which they can make an immediate impact. This will further reduce the most critical resource of an open source project, namely contributing developers.

When a project fails to meet release targets several times, the project as a whole, and its release manager in particular, loses credibility. The realization by developers, end users, and vendors that a project's release schedule is unpredictable could result in dropping adoption and support of the project altogether.

Regular, predictable releases help increase motivation for three main reasons. First, developers who contribute features or bug fixes know their contributions will be available to users within a fairly short time. Second, positive user feedback can reinvigorate developers. Finally, a more organized and predictable process may also prompt potential contributors to engage with the project as a regular release heartbeat is an indication of a project that delivers.

How to Adopt Time-Based Releases

On the basis of insights from our study, we identified the following steps for implementing time-based releases (see Figure 1).

Step 1. Assess the project's suitability
- Project characteristics
- Developer buy-in

Step 2. Determine the release interval
- Regularity and predictability
- The nature of the project and its users
- Commercial factors
- Cost and effort
- Network effects

Step 3. Implement the release schedule
- Identify dependencies
- Plan the schedule
- Avoid specific periods

FIGURE 1. Guidelines for implementing time-based releases.

Assess the Project's Suitability

Despite the various benefits, a time-based strategy may not be suitable for every project. Thus, the first step is to decide whether or not a project is suitable for a time-based release strategy.

Project characteristics. A project's characteristics help determine whether a time-based release approach is appropriate. It doesn't make much sense to release regularly if there has been little work done that would warrant a new release. Furthermore, changing the release strategy can cause significant disruption. Therefore, projects that don't have problems with release management shouldn't risk changing unless they see clear benefits. The need for change was clear in the GNOME project, as one participant described:

> *I guess GNOME was close to being a disaster. Everyone agreed that it had to change, and they were looking for a [solution].*

Developer buy-in. While a release manager has the formal authority to make decisions regarding releases, it's important that he or she doesn't alienate voluntary contributors by failing to consult with them on major changes, which could lead to them leaving a project. One release manager suggested that

> *releasing only works if all involved developers just trust [the release managers] to do it. If people think it won't happen anyway, they will just upload broken changes and then it won't happen.*

If a project has failed to meet targets for several years, introducing a time-based strategy won't convince developers straightaway that deadlines are now real. To build trust, contributors must gradually gain positive experience with the new process. A successful implementation of time-based releases also relies on introducing appropriate control structures, as one participant illustrated:

> *Simply declaring that you'd release every six months is not good enough. It's actually a whole set of rules that were implemented at the time that made it go again.*

However, establishing such control structures and getting them accepted by developers takes time. Furthermore, without enforcing these control mechanisms, a project might regress into its previous, unstructured process.

Determine the Release Interval

The next step is to choose an appropriate release interval. We identified five factors that affect the choice of interval.

Regularity and predictability. If the interval is too short, contributors will have insufficient time to integrate their work and test adequately. On the other hand, long release cycles lead to many changes, which could result in some of the problems we mentioned earlier. Furthermore, planning is much more difficult with long release cycles. As one release manager explained,

> *When you speak about release cycles of 1.5 years, it's much harder to make a forecast about what will happen, what it will look like.*

The nature of the project and its users. Different users have varying requirements, and the type of users also depends on the nature of the project. For example, the GNU Compiler Collection (GCC) is a development tool whose users are (necessarily) developers. Developers may often be interested in frequent releases that deliver "cutting edge" software. They'll also be able to cope with technical difficulties, whereas less tech-savvy users might prefer more rigorously tested software. Such users might lean toward a slower release cycle that doesn't require them to upgrade too frequently.

The nature of the project may influence the release interval as well. For example, both the Linux kernel and X.org include hardware drivers. Since new hardware appears on the market regularly, projects trying to support new hardware are dealing with a moving target. A project may choose a short release cycle to cope with this continuous need to deliver additional functionality. Or, it may split such functionality (such as hardware drivers) from the main software package. The X.org project adopted the latter solution.

> A short release interval might allow open source projects to compete more effectively with proprietary software products.

Commercial factors. Commercial interests could also affect the release interval. For example, many open source projects rely on commercial Linux vendors for wide distribution of their software. Although open source projects are freely accessible through the Internet, being included in a commercial distribution (such as Red Hat) can benefit open source projects. This will make them more widely distributed and help them gain a larger user base—a goal of any open source project.

A member of the OpenOffice.org community gave another example. The OpenOffice.org project is served by community members who write books on the software. A too-short release interval would negatively impact book sales:

> *It would be the death if you would really bring out a new release every three months with new features. They would not sell any books at all.*

The publicity factor is another related, important factor. Projects with a long release cycle may have a "big bang" release that may create a splash of publicity, but more frequent releases could lead to constant press coverage of the project's progress. This is important information for potential users because it suggests an actively maintained project.

Finally, a short release interval might allow open source projects to compete more effectively with proprietary software products—for example, Mozilla Firefox versus Microsoft Internet Explorer—that typically have much slower release cycles. As one participant said,

> *A fast release schedule gives substantial advantages over a competitor with a long cycle; for*

example, during their beta cycle, several new releases of your product will hit the market.

Cost and effort. Several cost factors are important for deciding on a release interval. First, old releases typically must be supported for a certain amount of time. A short release cycle will increase the maintenance burden, as one participant explained:

If you have to maintain lots of older releases, it creates a huge burden.

Furthermore, the preparation of releases may require significant effort in large projects. One member of the OpenOffice.org QA team commented,

There is lots of work associated with a release. I don't want to do a full release test every two months.

Interestingly, this comment was made several months before OpenOffice.org moved to a three-month release interval. After about a year, the project announced its intention to move to a six-month release cycle, for two reasons. First, users expressed a preference for less frequent delivery of new features. Second, the QA team found it difficult to thoroughly test releases within the relatively short cycle of three months.

Too-frequent releases also fragment a project's user base, making it harder to track outstanding issues. Feedback could become less relevant if many users remain with older versions that differ from the current development version. Besides additional effort for developers, end users may be adversely affected by additional effort related to upgrades (installation and configuration, the learning curve for a new version, and so on).

A final issue is the potential difficulty of making large (or radical) changes to the code base. However, these need not occur in the main development line. Significant features may comprise several releases in a separate branch, which can be merged once they're finished.

Network effects. A project can gain tremendous advantages if it synchronizes its release schedule with the schedules of other projects from which it can leverage benefits. For example, a key reason why the Plone project moved to a six-month time-based strategy was to align its development more closely with that of Zope, as Plone is built on top of the latter. A Plone release manager outlined how the implementation of a similar release strategy enabled the project to use Zope's newest features:

In order to stay up to date, we need to tie ourselves to a specific Zope release, and the way to do that is to have our release schedules synchronized.

Network effects can also be observed in the influence of vendors who contribute to open source projects and vendors who ship the project in distributions (such as Linux distributions). For example, at some point, there was some debate on the GCC mailing list as to whether to skip a certain version, since two distributors who were also main contributors to the project stated they had no intention to deploy that particular version.

The QA team found it difficult to thoroughly test releases within the relatively short cycle of three months.

Implement the Schedule

Finally, the third step defines activities to implement the release schedule.

Identifying dependencies. In large projects consisting of many components, much of the work can start only when other tasks have been completed. It's important to identify such dependencies when creating a schedule. Offering general advice on how to identify such dependencies is difficult as it relies on experience with, and the context of, each unique project. Nevertheless, a good starting point is to investigate a project's past problems. Then, institute clear deadlines so that developers working on a specific activity know in advance when a dependent task is due.

Although being overly strict in a volunteer-based project can be difficult, it's important to make clear that deadlines are firm. If a release manager doesn't enforce these deadlines, it's likely that volunteers will increasingly ignore them, leading to further delays. A release manager should aim to reduce dependencies where possible. One way to do this is by identifying fallback options. For example, if the new interface of a library isn't finished by a certain time, it should be possible to go back to the previous version of the library.

Planning the schedule. Developers may work on new features outside the main development branch and integrate the work once it's finished. In fact, some projects, including OpenOffice.org, develop most of their features in such a "branched" way. One participant clarified that

> *if you insist on doing all development work on [the main development branch], then yes, of course you have a problem because you need a certain amount of time to get anything done.*

This suggests that the structure of the development process has an important impact on the schedule. A regular release strategy requires the development tree to be fairly stable over time. Some practices we observed to keep the development tree stable are the use of branches for features that take significant effort, peer review, testing discipline, and reverting and postponing features.

Schedule planning should also take into account activities such as testing and translations (for multilingual projects).

Projects should establish and enforce a clear policy to ensure that milestones are achieved and deadlines are respected. By actively omitting or postponing features that aren't ready, a release manager asserts control and shows developers that deadlines are enforced. One practice to enforce this is the use of freezes (such as code freezes), which set clear deadlines for certain changes.

Avoiding specific periods. When deciding on a release schedule, projects should consider the availability of the voluntary contributors. While proprietary software vendors have defined holidays, many open source projects operate throughout the year. Although development might take place at any time, there are nevertheless restrictions as to when a release can be prepared. For example, the projects we studied actively avoided the Christmas period, during which many volunteers might not be available. One participant explained,

> *One shouldn't have the freeze of all packages while people are away on Christmas vacation. We cannot release exactly on Christmas or Easter where important people won't be available.*

Most projects in our study also have their own events, such as conferences, during which participants are unavailable, and hence a release at such moments should also be avoided. Other periods are less critical—evidence as to whether to avoid the summer break is much less conclusive. Although many volunteers might be unavailable during that period, the absence of different contributors is spread over several months, so this "balances out." In fact, some contributors, such as students, might even increase their involvement as they have more time available.

While a time-based release strategy offers several benefits, it's important to realize that a time-based release strategy doesn't necessarily benefit all projects. Our study has a number of limitations that could be addressed by further research. First, we can't draw any conclusions as to whether our findings apply to proprietary software development because different considerations, such as economic and business factors, affect decisions regarding a release strategy. Second, the increasing involvement of the software industry in open source projects over the last decade will introduce additional constraints and factors that may affect the suitability of a time-based release strategy for those projects, especially if these projects have a significant role in the release process. Finally, we should emphasize that the steps we identified for adopting a time-based strategy don't guarantee success. Adopting a time-based release approach is challenging, which we also found to be true for the projects in our study.

Acknowledgments

We thank the anonymous reviewers whose comments helped improve this article. This research was supported partly by Science Foundation Ireland under grant 10/CE/I1855 to Lero—The Irish Software Engineering Research Centre (www.lero.ie), Enterprise Ireland under grant IR/2013/0021 to ITEA2-SCALARE, and the Irish Research Council.

References

1. E.S. Raymond, *The Cathedral and the Bazaar*, O'Reilly, 2001.
2. H.K. Wright and D.E. Perry, "Release Engineering Practices and Pitfalls," *Proc. 34th Int'l Conf. Software Eng.*, 2012, pp. 1281–1284; doi:10.1109/ICSE.2012.6227099.
3. *Proc. 1st Int'l Workshop Release Eng.* (RELENG 13), B. Adams et al., eds., 2013; doi:10.1109/ICSE.2013.6606779.
4. N. Kerzazi and F. Khomh, "Factors Impacting Software Release Engineering: A Longitudinal Study," *Proc. 2nd Workshop Release Eng.*, 2014.
5. S. Phillips, G. Ruhe, and J. Sillito, "Information Needs for Integrating Decisions in the Release Process of Large-Scale Parallel Development," *Proc. 2012 ACM Computer-Supported Collaborative Work* (CSCW 12), 2012, pp. 1371–1380; doi:10.1145/2145204.2145408.
6. M. Steff, B. Russo, and G. Ruhe, "Evolution of Features and Their Dependencies—an Explorative Study in OSS," *Proc. 2012 ACM Int'l Symp. Empirical Software Eng. and Measurement*, 2012, pp. 111–114; doi:10.1145/2372251.2372270.

ABOUT THE AUTHORS

MARTIN MICHLMAYR is an open source community expert at Hewlett-Packard. Previously he was the project leader for the Debian project. His research interests include open source, quality, and release management. Michlmayr received a PhD in technology management from the University of Cambridge. Contact him at tbm@cyrius.com.

BRIAN FITZGERALD is chief scientist at Lero—The Irish Software Engineering Research Centre and holds the Frederick Krehbiel Chair in Innovation in Business and Technology at the University of Limerick. His research interests include open source software, inner source, crowdsourcing, and lean and agile methods. Fitzgerald received a PhD in computer science from the University of London. Contact him at bf@lero.ie.

KLAAS-JAN STOL is a research fellow at Lero—The Irish Software Engineering Research Centre. His research interests include open source software, inner source, and agile and lean methods. Stol received a PhD in software engineering from the University of Limerick. Contact him at klaas-jan.stol@lero.ie.

7. B. Fitzgerald, "The Transformation of Open Source Software," *MIS Q.*, vol. 30, no. 3, 2006, pp. 587–598.
8. J.M. Gonzalez-Barahona et al., "Understanding How Companies Interact with Free Software Communities," *IEEE Software*, vol. 30, no. 5, 2013, pp. 38–45; doi:10.1109/MS.2013.95.
9. J.M. Gonzalez-Barahona and G. Robles, "Trends in Free, Libre, Open Source Software Communities: From Volunteers to Companies," *it - Information Technology*, vol. 55, no. 5, 2013, pp. 173–180; doi:10.1515/itit.2013.1012.
10. K. Stol and B. Fitzgerald, "Inner Source—Adopting Open Source Development Practices within Organizations: A Tutorial," *IEEE Software*, preprint, 2 May 2014; doi:10.1109/MS.2014.77.
11. T. O'Reilly, "Lessons from Open Source Software Development," *Comm. ACM*, vol. 42, no. 4, 1999, pp. 33–37; doi:10.1145/299157.299164.
12. P.C. Rigby et al., "Contemporary Peer Review in Action: Lessons from Open Source Development," *IEEE Software*, vol. 29, no. 6, 2012, pp. 56–61; doi:10.1109/MS.2012.24.
13. K. Fogel, *Producing Open Source Software: How to Run a Successful Free Software Project*, O'Reilly, 2005.
14. M. Michlmayr and B. Fitzgerald, "Time-Based Release Management in Free and Open Source (FOSS) Projects," *Int'l J. Open Source Software and Processes*, vol. 4, no. 1, 2012, pp. 1–19; doi:10.4018/jossp.2012010101.
15. B. Rossi, B. Russo, and G. Succi, "Analysis of Open Source Software Development Iterations by Means of Burst Detection Techniques," *Proc. 2009 Int'l Conf. Open Source Systems*, 2009, pp. 83–93; doi:10.1007/978-3-642-02032-2_9.
16. A. Wiggins, J. Howison, and K. Crowston, "Heartbeat: Measuring Active User Base and Potential User Interest in FLOSS Projects," *Proc. 2009 Int'l Conf. Open Source Systems*, 2009, pp. 94–104; doi:10.1007/978-3-642-02032-2_10.
17. N. Choi, I. Chengular-Smith, and A. Whitmore, "Managing First Impressions of New Open Source Software Projects," *IEEE Software*, vol. 27, no. 6, 2010, pp. 73–77; doi:10.1109/MS.2010.26.

The Highways and Country Roads to Continuous Deployment

Marko Leppänen, Tampere University of Technology

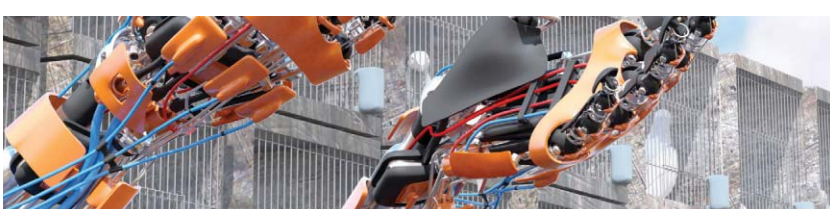
Simo Mäkinen and Max Pagels, University of Helsinki

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 prerelease versions 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 domain in which a company operated affected the flow of continuous deployment.

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.

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.

Achieving Reliable High-Frequency Releases in Cloud Environments

Liming Zhu, Donna Xu, An Binh Tran, Xiwei Xu, Len Bass, Ingo Weber, and Srini Dwarakanathan, NICTA

// Cloud applications with high-frequency releases often rely heavily on automated tools and cloud infrastructure APIs, both of which raise reliability issues. Experiments show that tradeoffs are also involved in the choice between heavily and lightly baked virtual-image approaches. //

CONTINUOUS DELIVERY is reducing release cycles from months to days or even hours. For example, Etsy.com had 4,004 releases into the production environment in six months, with an average of 20 releases per day and 10 commits per release.[1] Such high-frequency releases often rely on cloud infrastructure APIs and virtual machine (VM) images for initial provision, and then on deployment-related tools to complete the deployment or upgrade. However, this high frequency introduces reliability challenges.

In the past, developers often conducted infrequent, carefully monitored deployment or upgrades during scheduled downtime. Adopting cloud computing has given developers the opportunity to automate these tasks and moderately increase the release frequency. Previously, small reliability issues with the cloud infrastructure APIs or automated tools didn't pose a considerable threat because sufficient time existed to resolve the occasional issue. However, this is no longer true when—as our observations confirm—these APIs are called thousands of times a day and automated tools are used for true continuous delivery.

Applications in the cloud typically run on VMs, which are instances launched from a VM image, which typically takes one of two forms:

- *Heavily baked images*. The image includes all the software and most (if not all) of the configuration to run in an instance. (This form might also refer to heavily baking the immutable or phoenix servers, which aren't expected to change after booting, to prevent configuration drift.)
- *Lightly baked images*. The image contains only some of the necessary software, such as the OS and middleware. Each instance must load the remainder of the necessary software after being launched.

Considerable debate remains around how much baking is necessary, which further contributes to the reliability issues.

Here, we compare these two philosophies and report on specific reliability issues and tradeoffs. We identify major contributing factors at both the cloud-infrastructure and deployment-tool levels. Our general finding is that the more external resources you involve in a deployment—and the more you ask of those resources—the more likely you'll experience errors or delays. We propose various error-handling

0740-7459/15/$31.00 © 2015 IEEE

practices and ways to resolve the issues, including cloud API wrappers and intermediary outcome validation to detect errors much earlier.

Motivating Example: Rolling Upgrades

Assume an application running in the cloud consists of a collection of VM instances, instantiated from a few different VM images. A new machine image representing a new release for one image (VM_R) is available for deployment. The current version of VM_R is V_A; the goal is to replace the N instances currently executing V_A with N instances executing the new version V_B. A further goal is to do this replacement while providing the same service level to VM_R clients. That is, at any point during replacement, at least N instances running some VM_R version should be available for service. One way to achieve this is with a *rolling upgrade*.[2]

> The more external resources you involve in a deployment, the more likely you'll experience errors or delays.

A rolling upgrade takes out of service a small number of k instances at a time currently running V_A and replaces them with k instances running V_B. This technique can meet the requirement for N instances running some version of VM_R by creating the same number (k) of additional instances running V_B as are simultaneously being upgraded—thus overprovisioning for the upgrade. The replacement usually takes on the order of minutes. If k is set to 10 percent of N, a rolling upgrade can usually take less than an hour—even for hundreds or thousands of instances. Rolling upgrades are popular; their virtue is that they require only a few additional instances.

During an upgrade, three categories of failures can occur. *Provisioning failures* occur during replacement—specifically, when one upgrade step produces incorrect results. Here, we examine these failures by comparing reliability issues during provisioning using heavily and lightly baked images.

Logical failures are related to the application being upgraded, such as version incompatibility or inter-instance dependencies. These failures are application-specific; we don't discuss them here.

Instance failures are a normal occurrence in the cloud. They can be due to failure of the underlying physical machine, the network, or a (networked) disk. Because they're not specific to deployment and upgrades, we don't discuss them here. They might occur within or outside the rolling-upgrade period and can be dealt with using traditional fault-tolerance mechanisms.

Experiments and Observations

We performed rolling upgrades using Amazon Web Services (AWS) Elastic Compute Cloud (EC2) and OpsWorks.

OpsWorks is Amazon's automated DevOps (development and operations) tool, which integrates with Chef configuration management to fully provision an application service. In OpsWorks, an application service has a set of life-cycle events associated with custom-built Chef recipes. By calling operations from the OpsWorks API and other EC2 APIs, we can replace a configurable number of old application service instances with new ones. We're dealing with both cloud infrastructure APIs (EC2 APIs) and deployment software (OpsWorks, Chef tools, APIs, and so on). Both types contribute to the upgrade's reliability.

We configured the rolling upgrade to support both the heavily and lightly baked approaches. Heavily baked upgrades used a custom Amazon Machine Image (AMI) with built-in recipes, which largely performed OpsWorks-related agent setups and simple default configurations. Lightly baked upgrades used a basic AMI with more complex custom Chef recipes, which performed more customized actions for installing additional software. These upgrades required more actions after instantiation and relied much more on the deployment software (OpsWorks and Chef) than the heavily baked upgrades.

From an abstract viewpoint, the two approaches perform the same task: a rolling upgrade of an application. In our experiments, the application stack was a standard three-tier (Web, application, and database server) deployment of a content management system. We upgraded an application server (Tomcat) because it was located centrally in the stack with complex dependencies. We performed rolling upgrades to a cluster of 72 servers, with k set to 3, and recorded timings of different phases and reliability problems in each case.

Figure 1 shows that, overall, lightly baked upgrades were less reliable than heavily baked upgrades.

We recorded the time distribution for the four upgrade phases:

- *Stopping or terminating*. The API operation for stopping or terminating an instance was called.
- *Pending*. OpsWorks is waiting for a new EC2 instance to start.
- *Booting*. An EC2 instance is booting.
- *Running setup*. OpsWorks is running Chef recipes.

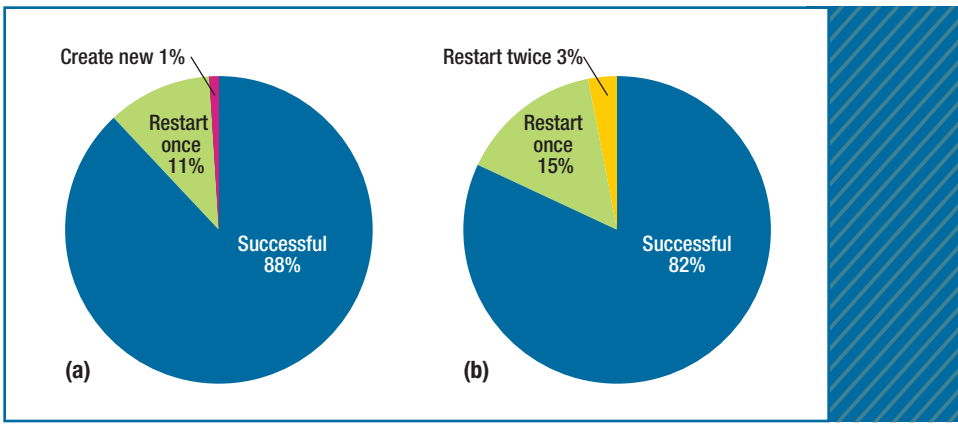


FIGURE 1. Success rates for (a) heavily and (b) lightly baked upgrades. Heavily baked upgrades were generally more reliable.

Figure 2 shows the time distribution. We performed statistical tests and observed statistical significance in the timing profiles and all other reported timing differences. We have two key observations. First, lightly baked upgrades usually completed faster, typically in 4 to 6.5 min. Most heavily baked upgrades took 8 to 10.5 min.

Second, lightly baked upgrades had a broader distribution than heavily baked upgrades. The lightly baked upgrades' completion times could be significantly longer; their distribution had considerably longer tails (see the sidebar).

These two observations demonstrate that heavily baked upgrades were stable but usually took longer.

Figure 3 shows the time distributions for the four upgrade phases. For stopping and pending, the distributions for lightly and heavily baked upgrades are relatively close on each time interval. For booting, heavily baked upgrades take considerably longer. What exactly OpsWorks does with an instance in this phase isn't clear: the documentation states only

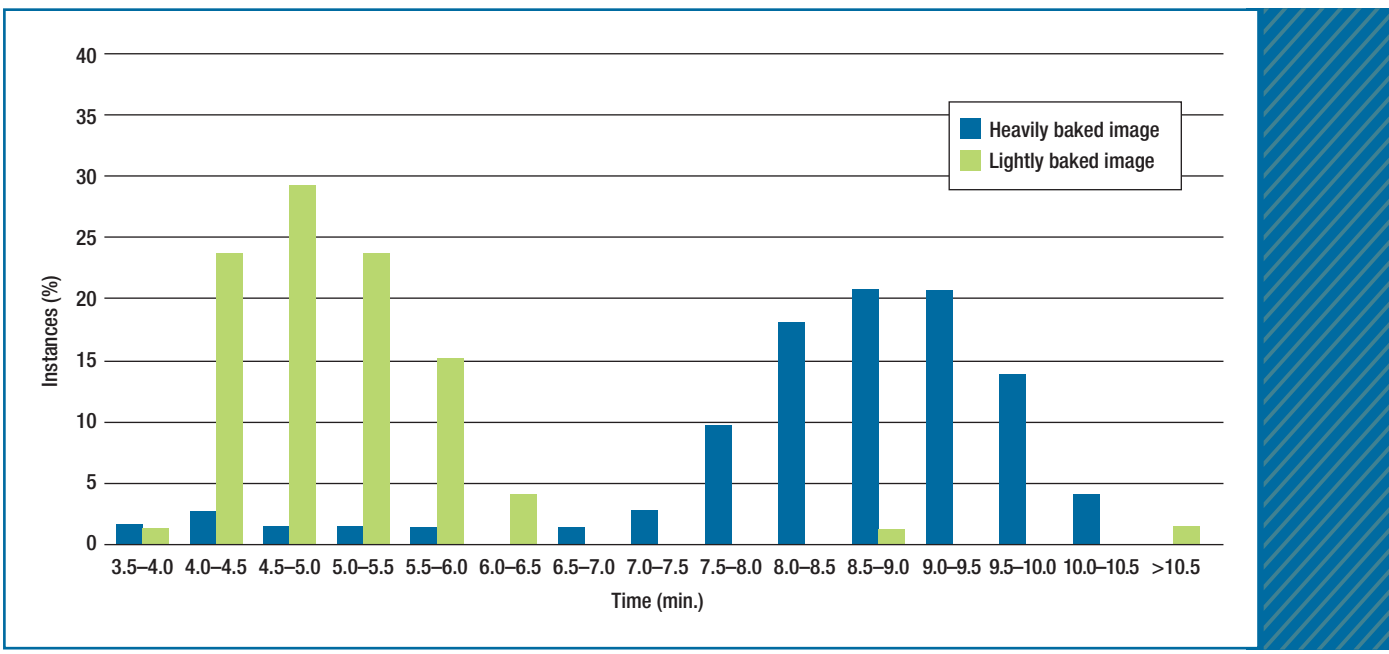


FIGURE 2. The time distribution for upgrading one instance. Lightly baked upgrades usually completed faster, with most instances taking 4 to 6.5 min.

THE LONG TAIL

A probability distribution has a long tail if a larger proportion of the population resides in the tail than there would be under a normal distribution. In other words, the 68–95–99.7 rule states that the expected percentages of population lie within one, two, or three standard deviations from the mean in a normal distribution, respectively. If the proportion is larger than these percentages, the tail is long. This often has considerable implications for large-scale applications;[1] such implications extend over the large-scale applications themselves onto operations on these applications.[2]

References

1. J. Dean and L.A. Barroso, "The Tail at Scale," *Comm. ACM*, vol. 56, no. 2, 2013, pp. 74–80.
2. X. Xu et al., "POD-Diagnosis: Error Diagnosis of Sporadic Operations on Cloud Applications," *Proc. 44th Ann. Int'l Conf. Dependable Systems and Networks*, 2014, pp. 1–12.

that the OpsWorks agent is installed. Users reported various problems in this phase on the support forum. Lightly baked upgrades (including booting and running setup) can take longer, which might be attributable to unreliable on-demand service installation and configuration.

To better understand the unreliability's sources, we compared the AWS-related API calls and Chef recipe size between the two approaches. To capture the API calls, we used Amazon CloudTrail, which can log all AWS API calls. Lightly baked upgrades triggered 40 EC2 API calls, whereas heavily baked upgrades triggered 37. This difference isn't

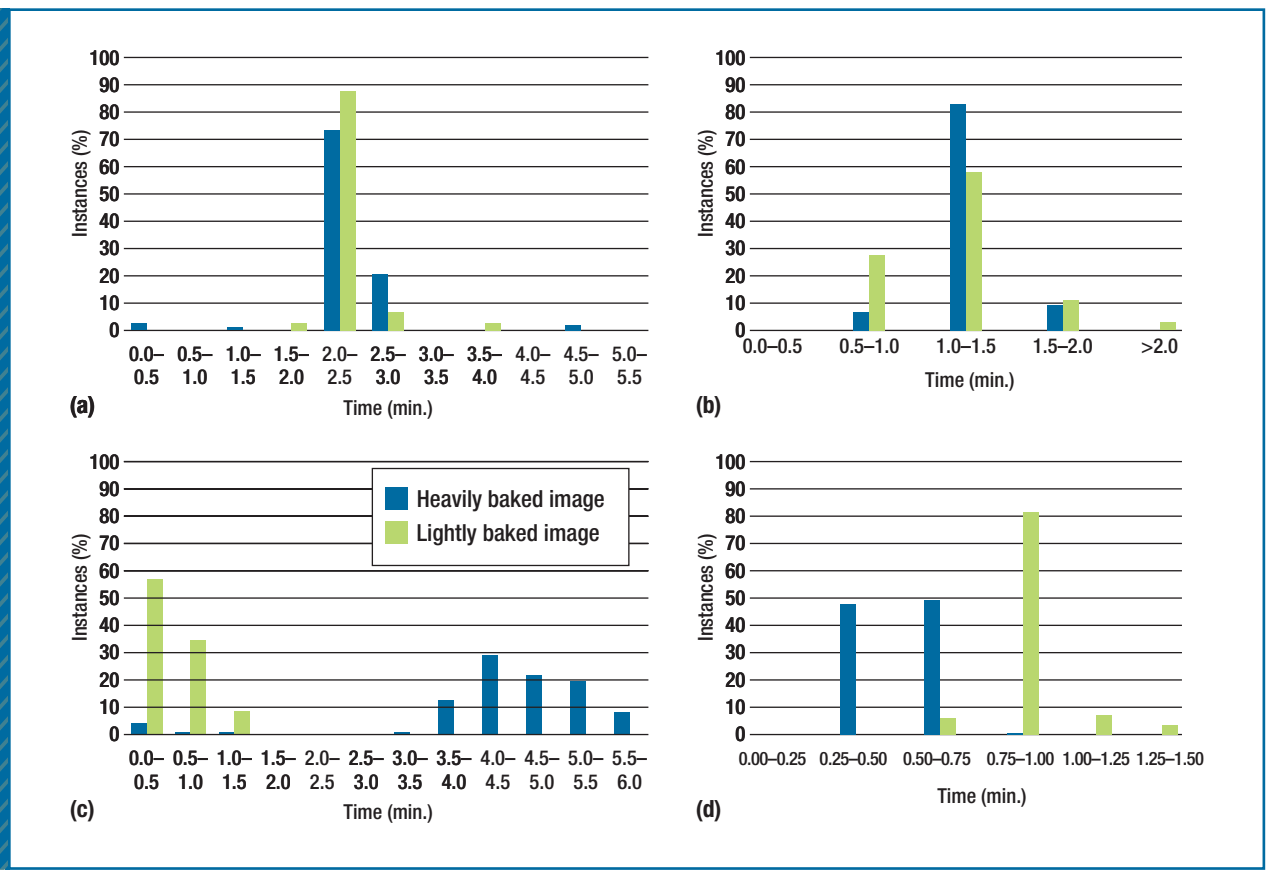


FIGURE 3. Time spent per status. (a) Stopping. (b) Pending. (c) Booting. (d) Running setup. For booting, the heavily baked upgrades took considerably longer; OpsWorks users also reported various problems during this phase in the user support forums.

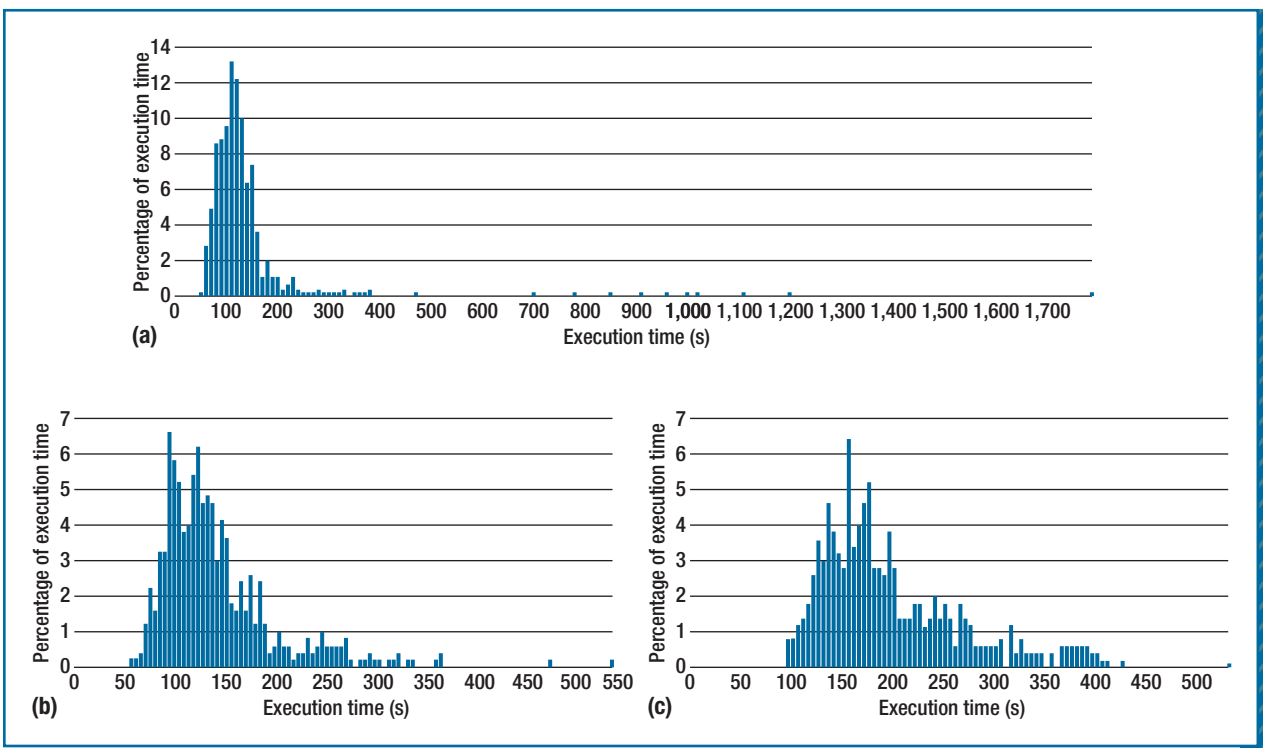


FIGURE 4. The execution time for the major steps of lightly baked upgrades. (a) **update_custom_cookbooks**. (b) **execute_recipes**—uninstall Tomcat 6. (c) **execute_recipes**—install Tomcat 7. All three steps clearly showed a long tail.

material in terms of the AWS basic infrastructure contributing to unreliability. For the Chef recipes, heavily baked upgrades had 69 OpsWorks API calls per upgrade, whereas lightly baked upgrades had 142.

Many of the errors came from Chef and the OpsWorks agent that acted as a wrapper around the Chef agent. We thus suspect that the additional Chef- or OpsWorks-related actions in the lightly baked upgrades were the major contributors to slowness, the long tail, and unreliability. Next, as we describe later, we investigated the factors and tried to establish whether many of the additional actions followed the long-tail characteristics in completion time.

During lightly baked upgrades, Chef's actions constituted three main steps:

1. Execute the **update_custom_cookbooks** deployment action. This step updated new custom Chef cookbooks from an external Git repository to the local instance cache over the network. In our case, the recipes were about the installation of a new Tomcat version (from version 6 to 7).
2. Execute the uninstallation-related **execute_recipes** deployment action. This ran the **tomcat::uninstall** Chef recipe to remove the old Tomcat version.
3. Execute the installation-related **execute_recipes** deployment action. This ran the **tomcat::setup** and **tomcat::configure** Chef recipes to install and configure the new Tomcat version.

As Figure 4 shows, all three steps clearly showed a long tail. On further examining the time stamps in logs, we observed two major contributors to the long tails.

The first surprise is the delay in action commands being acknowledged, executed, and reported, which contributed approximately 28 percent to the overall long-tail characteristics. Figure 5 shows the three delay types happening during each command's execution (excluding the execution itself):

- The delay from when the command was created until AWS acknowledged it for execution (labeled A in Figure 5). In our observations, this delay was large (from 10 to 300 seconds).
- A small delay from when AWS reported to have acknowledged

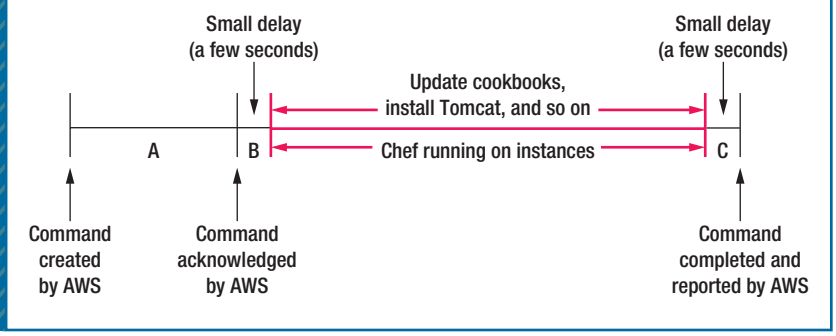


FIGURE 5. The three types of delays during each command's execution. Label A indicates the delay from when the command was created until AWS acknowledged it for execution. B indicates the delay from when AWS reported to have acknowledged the command and Chef actually started running on the instance. C indicates the delay from when the Chef run completed on the instance until AWS reported that the command completed.

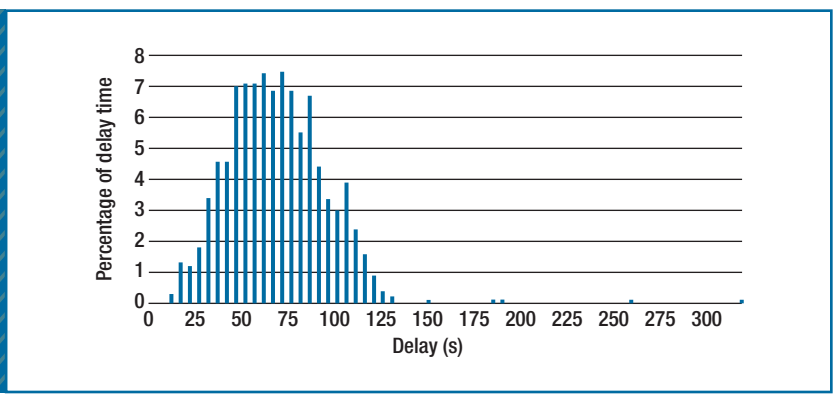


FIGURE 6. The total delay in command processing and reporting. This takes into account the three delays in Figure 5.

the command and Chef actually started running on the instance (we got this information from the Chef log). This delay is labeled B in Figure 5. It might have been caused by API calls or a delay in the OpsWorks agent triggering Chef on the instance. It was a few seconds at most.

- A small delay from when the Chef run completed on the instance until AWS reported that the command completed (labeled C in Figure 5). The delay time and reasons were similar to the previous small delay.

Figure 6 shows the time distribution for all three delays.

The second contributor to the delays stemmed from external resources—mostly, the software repository and dependency servers—which made up approximately 70 percent of the overall long-tail distribution. Downloading software from an external repository for installation or even resolving dependencies before removing an old version can have considerable long-tail characteristics. The remaining actions, such as configuring and updating cookbooks from local cache, show no such characteristics. Common industry practices use a local mirror server or some redundancy to relieve the issues around downloading from repositories.

One possibility is that the delays were due to Chef's somewhat unpredictable pull model, in which the configuration converges periodically toward the desired state. However, for OpsWorks and our rolling upgrade, this was transformed more into a push model: different types of Chef recipes were set to converge when triggered by specific conditions. For example, Chef recipes in OpsWorks' "configure" life-cycle event ran on all instances every time an instance in the OpsWorks stack entered or left the online state (for example, to start, stop, or terminate). So, Chef's pull model wasn't a major factor in this setup.

Discussion and Threats to Validity

This study has some obvious limitations. First, it built on AWS using OpsWorks and Chef, without testing other platforms.

Second, our comparison is limited: we didn't consider image preparation time, which might differ considerably between the heavily and lightly baked approaches. In the past, images were often prepared by starting an instance, installing the required software, and resealing it as an image before provisioning. In that process, time and reliability were major concerns.

However, the improved current practice is to use a dedicated *baking*

instance[3] that modifies a mounted image directly, considerably speeding up preparation. The process is also more reliable because the image is never started with on-demand configuration to get the required software. We believe the time and reliability issues are largely resolved in practice.

Third, 72 servers isn't a huge setup. We believe the outliers in the tail are proportional to the total number of servers upgraded in larger settings. But the outliers' effect on the total upgrade completion time might also depend on a rolling upgrade's granularity.

Only a few related approaches exist in this area. When a release process is automated through scripts, error-handling mechanisms in the scripting or high-level languages can detect and react to errors and reliability issues through exception handling. For example, error handlers in Netflix's Asgard (https://github.com/Netflix/asgard) and Chef (www.getchef.com/chef) react to detected errors. These exception-handing mechanisms are best suited for single-language environments, but continuous delivery often must deal with different types of error responses from different systems. Also, exception handling has only local information rather than global visibility when an exception is caught. So, external validation checks based on more global information are useful.

Our Proposed Solution

We now introduce some early solutions to the problems we've identified; our primary goal is to shorten the long tail in some operations.

To deal with the reliability issues, we incorporate fail fast, retry, and alternative actions in the rolling-upgrade tools. In our rolling upgrade using AWS OpsWorks, we implemented the following four error detection and handling mechanisms, which considerably reduced the reliability issues.

First, our system actively tracks each instance's status through the life cycle and the time spent in each life-cycle stage. The information is then used by approaches that we describe later.

> The reliability problems of deployment infrastructure and tools are considerable under the lightly baked approach.

Second, we implement asynchronous upgrades. For a rolling-upgrade granularity of k ($k > 1$), we don't wait until each wave is finished before starting the next wave. When a single instance has been upgraded and is online again, another instance is upgraded straightaway. The granularity then ensures that k servers are upgraded concurrently at any point in time. This also prevents recurring errors in an instance from blocking the whole upgrade. Should the problem recur on all subsequently started instances, the whole upgrade will fail-stop or be rolled back.

Third, we use time-outs specific to each status to fail fast. We collected historical data for upgrades and use the 95th percentile as the default (but configurable) time-out.

Finally, we provide stop–restart, replace, deploy without restart, and direct triggering of life-cycle events as alternatives for many actions.

In addition, OpsWorks heavily builds on Chef and, as we've seen, contributes considerably to unreliability. So, we implemented mini-test-based validation (https://github.com/calavera/minitest-chef-handler) of intermediary outcomes for the Chef portion. Previously, these tests were used during development; we're using them during production runs to detect errors early. These tests go beyond what Chef error reports or logs are reporting. They validate the expected final outcomes—not just inputs to a Chef execution or the execution itself.

Finally, for EC2 API reliability issues, we analyzed their characteristics and implemented an API wrapper to solve the problem.[4]

As our investigation clearly shows, the reliability problems of deployment infrastructure and tools—excluding script and user errors—are considerable under the lightly baked approach and exhibit long-tail characteristics at scale. Key contributing factors here include infrastructure API reliability, command processing and reporting delays, and dealing with external resources. The mechanisms we described in this article are good strategies to alleviate these problems. Furthermore, we're developing Process-Oriented Dependability, a framework that works with existing deployment tools by analyzing the logs they produce.[5]

The heavily baked approach can incur considerable preparation overhead because even minor changes

warranting a release require preparing a complex image. The numerous images that result must be stored and managed, creating "image sprawl." Also, an application might comprise many different images that must correspond to each other in some way. This monolithic approach often requires considerable coordination, thus delaying deployment.

Acknowledgments

The Australian government's Department of Communications and the Australian Research Council's ICT Centre of Excellence Program fund NICTA.

References

1. W. Stuckey, "Managing Experimentation in a Continuously Deployed Environment," 2013; www.slideshare.net/InfoQ/managing-experimentation-in-a-continuously-deployed-environment.
2. *Asgard*, Netflix, 2014; https://github.com/Netflix/asgard.
3. *Aminator*, Netflix, 2013; https://github.com/Netflix/aminator.
4. Q. Lu et al., "Mechanisms and Architectures for Tail-Tolerant System Operations in Cloud," *Proc. 6th Usenix Workshop Hot Topics in Cloud Computing*, 2014; www.usenix.org/conference/hotcloud14/workshop-program/presentation/lu.
5. X. Xu et al., "POD-Diagnosis: Error Diagnosis of Sporadic Operations on Cloud Applications," *Proc. 44th Ann. Int'l Conf. Dependable Systems and Networks*, 2014, pp. 1–12.

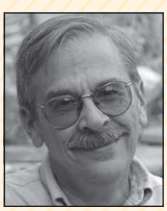
ABOUT THE AUTHORS

LIMING ZHU is a research group leader and principal researcher at NICTA, and a conjoint lecturer at the University of New South Wales and University of Sydney. His research interests include software architecture and dependable systems. Zhu received a PhD in software engineering from the University of New South Wales. Contact him at liming.zhu@nicta.com.au.

DONNA XU is a vacation student at the Commonwealth Scientific and Industrial Research Organisation. Her research interests include cloud computing, dependability, and big-data processing. Xu received a BCST (Honors) in computer science from the University of Sydney. Contact her at donna.xu@nicta.com.au.

AN BINH TRAN is a research assistant at NICTA. His research interests include development operations, cloud computing, dependability, and big data. Tran received a BSc (Honors) in computer science from the University of New South Wales. Contact him at anbinh.tran@nicta.com.au.

XIWEI XU is a researcher at NICTA. Her research interests include dependability, cloud computing, development operations, and big data, as well as software architecture, business processes, and service computing. Xu received a PhD in software engineering from the University of New South Wales. Contact her at xiwei.xu@nicta.com.au.



LEN BASS is a senior principal researcher at NICTA. His research interests include operating systems, database management systems, user interface software, software architecture, product line systems, and computer operations. He's coauthor of *DevOps: A Software Architect's Perspective* (Addison-Wesley, 2015), *Software Architecture in Practice* (Addison-Wesley, 2012), and *Documenting Software Architectures: Views and Beyond* (Addison-Wesley, 2010). Bass received a PhD in computer science from Purdue University. Contact him at len.bass@nicta.com.au.

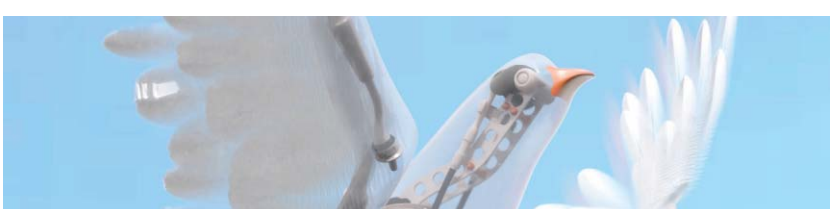
INGO WEBER is a senior researcher in the Software Systems Research Group at NICTA and an adjunct senior lecturer in computer science and engineering at the University of New South Wales. His research interests include cloud computing, development and operations, business process management, and AI. He's coauthor of *DevOps: A Software Architect's Perspective* (Addison-Wesley, 2015). Weber received a PhD in computer science from the University of Karlsruhe. Contact him at ingo.weber@nicta.com.au.

SRINI DWARAKANATHAN is a research intern at NICTA. His research interests include distributed systems, cloud computing, high availability, and software-defined networking. Dwarakanathan received an MS in computer and information science from the University of Pennsylvania. Contact him at srini.nathan@nicta.com.au.

Release Stabilization on Linux and Chrome

Md Tajmilur Rahman and Peter C. Rigby, Concordia University

// *An empirical study of release stabilization for Linux and Chrome found that few changes were reverted, small teams controlled the stabilization effort and did much of the rework, and despite using rapid release, some rushed changes still occurred before stabilization.* //



LARGE SOFTWARE PROJECTS make thousands of changes between releases. During development, new features and other major changes are implemented. Because relatively few developers and end users have used the new changes, those changes can destabilize the overall software system. A release engineer selects and stabilizes the changes before the system is released to a large user base.

We quantified the time and effort expended in the release stabilization of two large successful projects: the Linux kernel and Google Chrome. You can use our measurement tools (https://github.com/tajmilur-rahman/measurements) to compare your projects with Linux and Chrome regarding the questions covered in this article. To use the tools, all you need is a Git repository that contains tags indicating the start of release stabilization and the final release. We're willing to help you make these measurements and hope that such grounded empirical findings will help transform software development into an engineering discipline.

How Rapid Are Your Releases?

In Linux development's early days, releases sometimes occurred more than once a day, prompting Eric Raymond's mantra of "Release early. Release often."[1] This trend has continued, with many projects adopting increasingly shorter release intervals.[2] For example, Google Plus can release new changes in 36 hours,[3] and Facebook.com releases twice a day on weekdays.[4] Firefox and Chrome operate on six-week release cycles.[2,5]

To quantify the time and effort for release stabilization, we used the Linux and Chrome process documents' definitions of the development and stabilization branches.[5,6] Figures 1 and 2 show these branches. The stabilization and release tags in Git let us traverse the Git directed acyclic graph and identify on which branch a change was made. We extracted *churn*—the number of lines of code added and removed per commit—from the Git version history. We used the Git author field, not the committer field, to credit work to developers. For more details on our extraction process, see our previous paper.[7]

Figure 1 represents the Linux release process. Linux uses a flexible time-based release schedule, which consists of a merge window and stabilization period.[6] The merge window opens to let developers merge changes into the stabilization mainline. The window is open for only two weeks, with a standard deviation of two days. After the window closes, the first release candidate (rc1) will indicate the start of release stabilization. During stabilization, only fixes to regressions and isolated changes, such as device drivers, are merged. New release candidates will be created as regressions are found and fixed.

0740-7459/15/$31.00 © 2015 IEEE

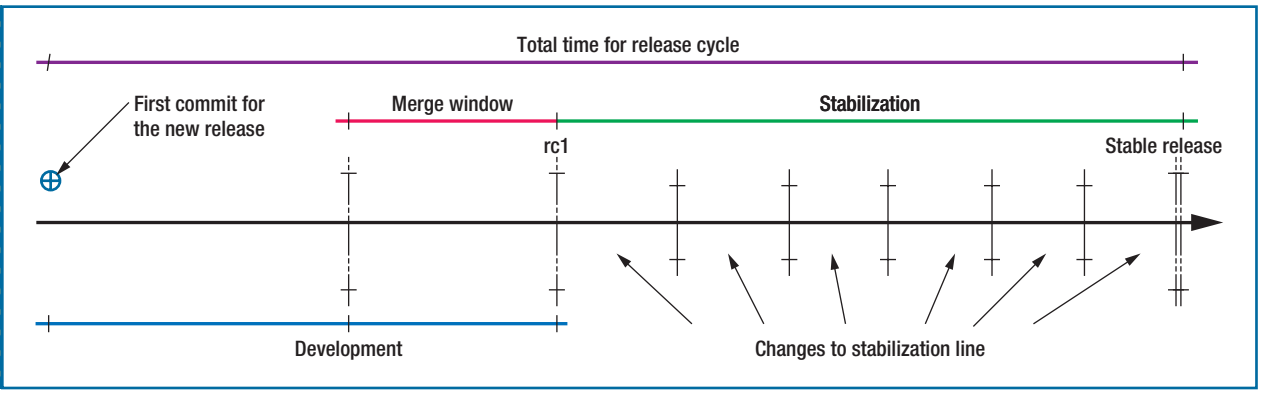


FIGURE 1. The Linux release process. Development of subsequent releases occurs in parallel with a release's stabilization. The two stages join during the merge window, where new development moves onto the stabilization mainline. rc1 is the first release candidate.

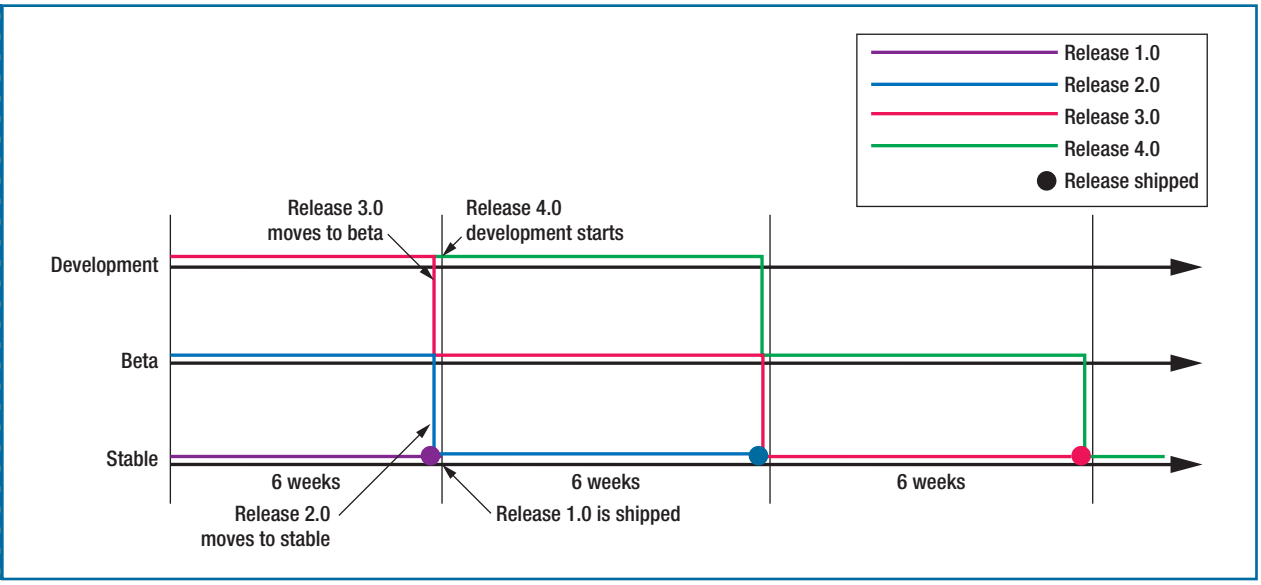


FIGURE 2. The Chrome release process. Development occurs in parallel with two stabilization branches—beta and stable. At six-week intervals, code on each branch moves to the subsequent branch—for example, development moves to beta.

We found that, on average, there were six release candidates before the final public release. The time period for stabilizing a release continued until no important regressions were outstanding. Stabilization took on average 62 days (represented by the horizontal line in Figure 3), with a standard deviation of 10 days, a minimum of 45 days, and a maximum of 93 days. Since release 2.6.31, release stabilization has become more regular. Figure 3 shows the variations in the Linux release cycle.

Chrome's release process consists of three channels: development, beta, and stable. At six-week intervals, the code transitions to the subsequent channel.[5] For example, Figure 2 shows that when development began on release 4, release 3 moved to the beta channel, release 2 moved to the stable channel, and release 1 was published as a final production release. In our data extraction scripts, we can identify on which channel a commit was made, on the basis of its version number. In this article, we don't differentiate between the beta and stable channels because both are related to release stabilization.

Release stabilization took an average of 91 days, with a standard deviation of 11 days, a minimum of 56 days, and a maximum of 149

days. Figure 3 shows the variations in Chrome's release cycle; the horizontal line shows the ideal 12-week stabilization period. Immediately after Chrome adopted a rapid release cycle, release times varied significantly, with some releases taking substantially longer than 12 weeks. Recent releases have become much more regular.

How Much Effort Do You Expend in Stabilizing a Release?

To get a sense of the effort involved in developing and releasing Linux and Chrome, we used three basic measures: the number of commits, the churn, and the number of people working on the development versus stabilization branches.

For Linux, of the 381K commits made to kernel source files between

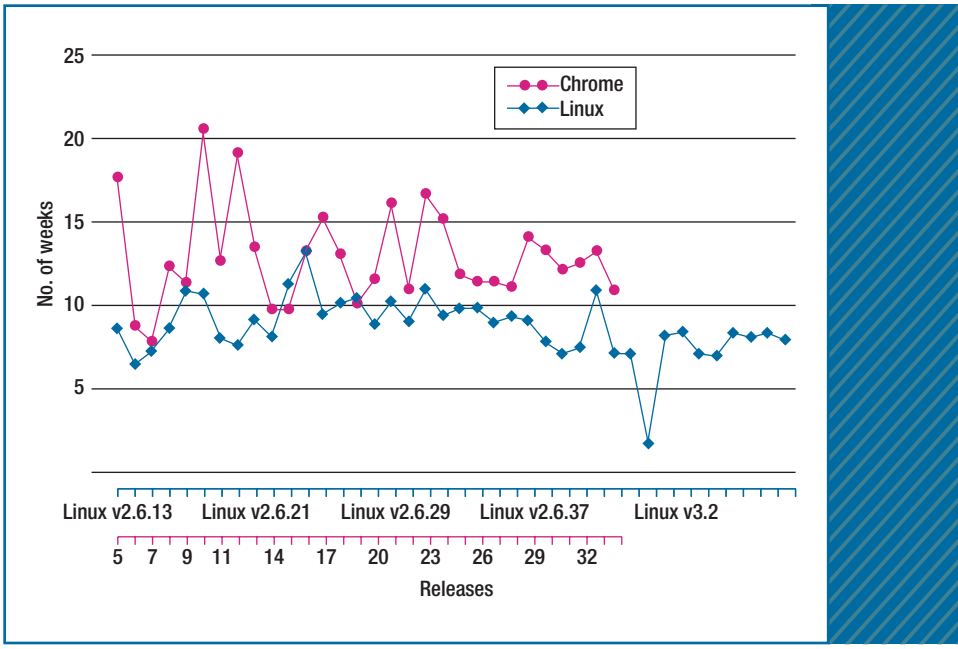


FIGURE 3. The length of stabilization and development periods for Linux and Chrome. Release stabilization varies by approximately 10 days and has become more regular.

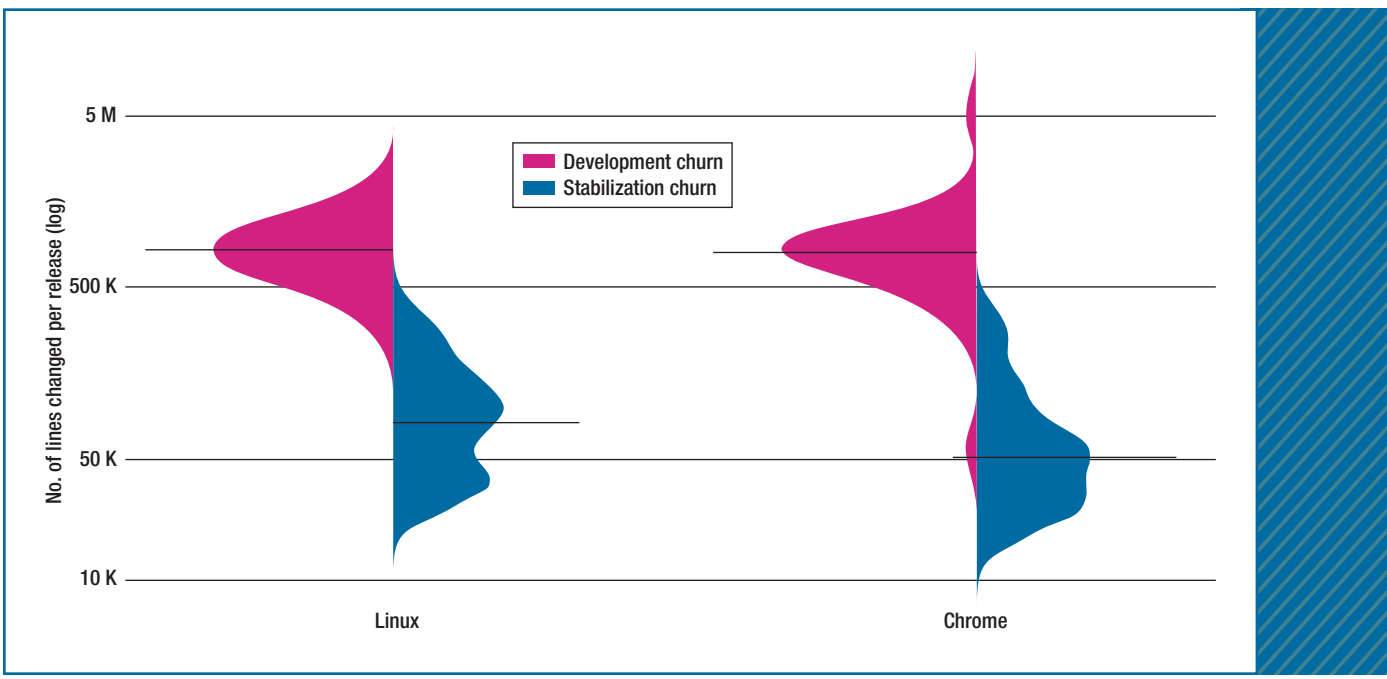


FIGURE 4. Churned LOC per release for development and stabilization. Most changes occurred during development, with only 9 percent of the lines changed during stabilization for Linux and 7 percent for Chrome. Beanplots show the distribution density for multiple samples along the *y*-axis (rather than more commonly along the *x*-axis) to enable easy visual comparison. In these beanplots, a horizontal line represents the median.

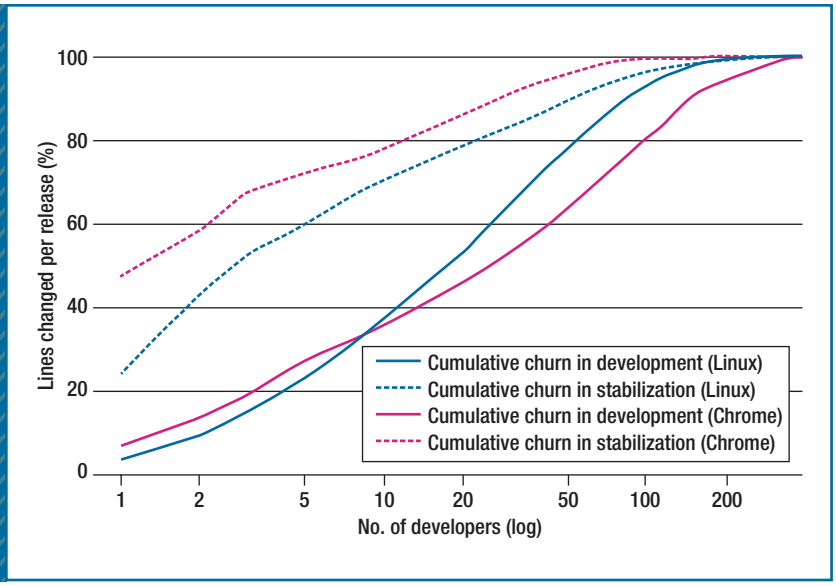


FIGURE 5. The cumulative distribution of developer contributions. A very small group of developers controlled the stabilization of a release: 23 developers for Linux and 10 for Chrome.

2005 and 2013, 77 percent were during development and 23 percent were during stabilization. As Figure 4 shows, the median development churn per release was 834 KLOC, compared to the stabilization churn of 83 KLOC. A Wilcoxon test showed that this difference was statistically significant, with $p << 0.001$. In the median case, 91 percent of the lines changed for a release were during development, with a ratio of 105 lines churned per commit, whereas 9 percent of the lines changed during stabilization, with 41 lines churned per commit. Linux release engineers tended to make small changes during stabilization.

For Chrome, of the 164K commits made to source files between 2008 and 2014, 85 percent were during development and 15 percent were during stabilization. The median development churn per release was 808 KLOC, compared to the stabilization churn of 51 KLOC (see Figure 4). A Wilcoxon test showed that this difference was statistically significant, with $p << 0.001$. In the median case, 93 percent of the lines changed for a release were during development, with a ratio of 11 lines changed per commit, whereas 7 percent of the lines were changed during stabilization, with 165 lines churned per commit. Chrome release engineers tended to make large changes during stabilization.

Figure 5 shows the distribution of developer contributions. For Linux, 10,000 developers contributed; however, 55 developers performed 80 percent of the development, and 23 developers performed 80 percent of the stabilization. For Chrome, 98 developers made 80 percent of the changes during development, and 10 developers made 80 percent of the changes during stabilization.

This result is similar to Audris Mockus and his colleagues' finding that the Apache httpd server had a core group of 15 developers who wrote 80 percent of the code.[8] Linux is a much larger project; 23 developers controlled stabilization. Mockus and his colleagues noted that as a system grows (for example, Mozilla), its management requires more complex mechanisms. The Mozilla project had 13 release engineers.[9] To integrate the development effort from the larger group of 55 developers that accounted for 80 percent of the development effort, Linux used a chain of trust to pass changes from less trusted developers to the trusted stabilization mainline that Linus Torvalds controls and makes releases from.[6] Stabilization occupies most of Torvalds's time and clearly represents large contributions from other core developers.

Do You Stabilize Your Own Code during a Release?

DevOps combines operational work, including release engineering, with development work. One example of a DevOps combination is requiring developers, instead of integrators, to fix their own code during stabilization. We find that much of the stabilization effort is still on the release engineers' shoulders.

The Linux Kernel has a policy that "the original developer should continue to take responsibility for the code [they contribute]."[6] Chrome also has this expectation.[5] We expect developers who modify files during development to fix any problems with those files that arise during stabilization.

Of the files that were modified in both periods, we measured the proportion of those that the original developer modified to those that other developers and integrators modified. In the median case per release,

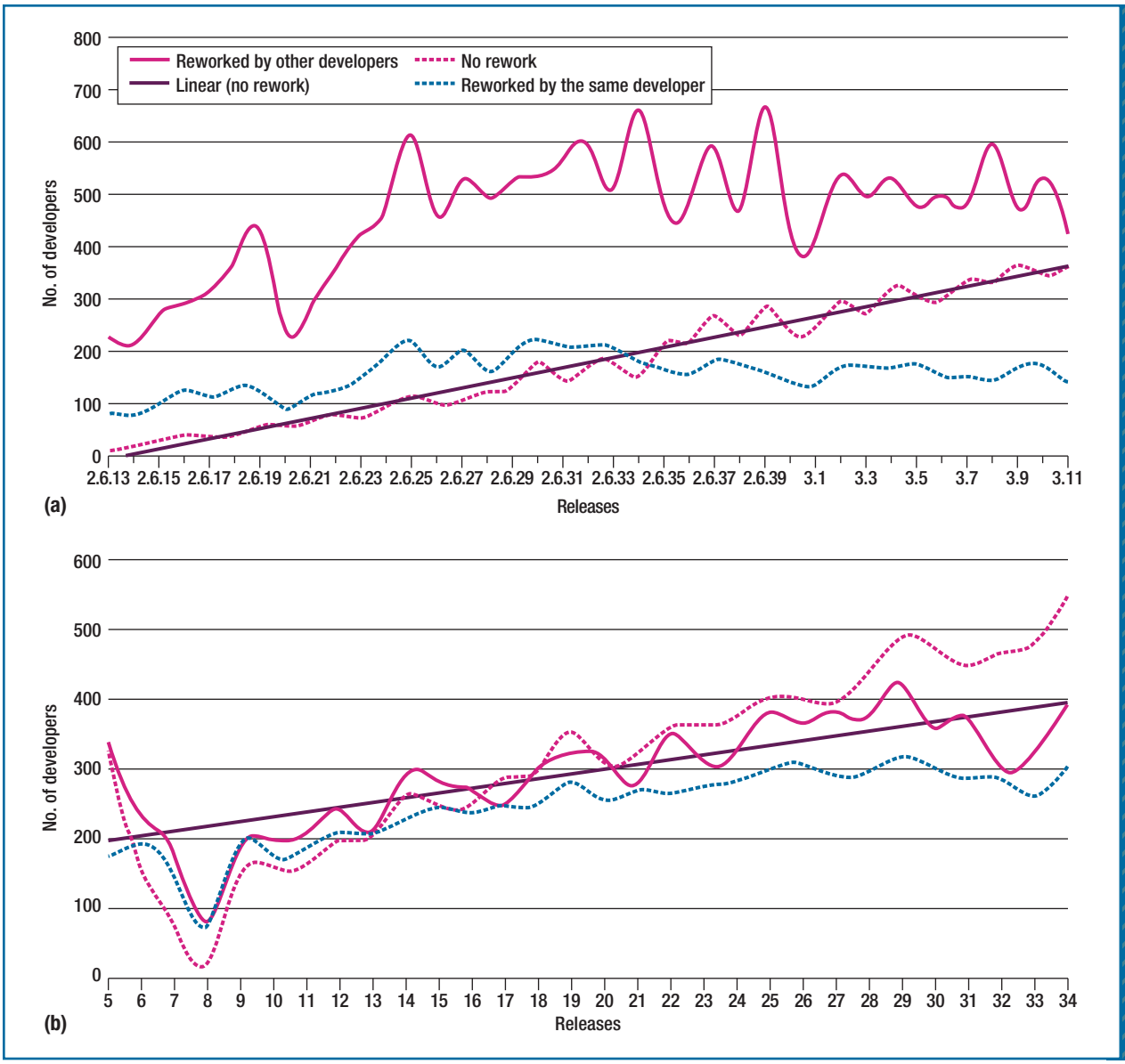


FIGURE 6. The number of (a) Linux and (b) Chrome developers reworking files during stabilization. Many developers had their files modified by another developer during stabilization.

- 161 Linux and 258 Chrome developers modified files during stabilization that they had changed during development,
- 480 Linux and 307 Chrome developers had their files modified by other developers during stabilization, and
- 171 Linux and 322 Chrome developers had changes requiring no modification during stabilization.

These sets of developers weren't mutually exclusive. Figure 6 depicts this situation for Linux and Chrome. From these numbers, it would appear that many developers didn't take on the responsibility to fix their bugs for a release. Because the number of developers making changes was much larger than the core group of developers, many of these changes were likely made by transient

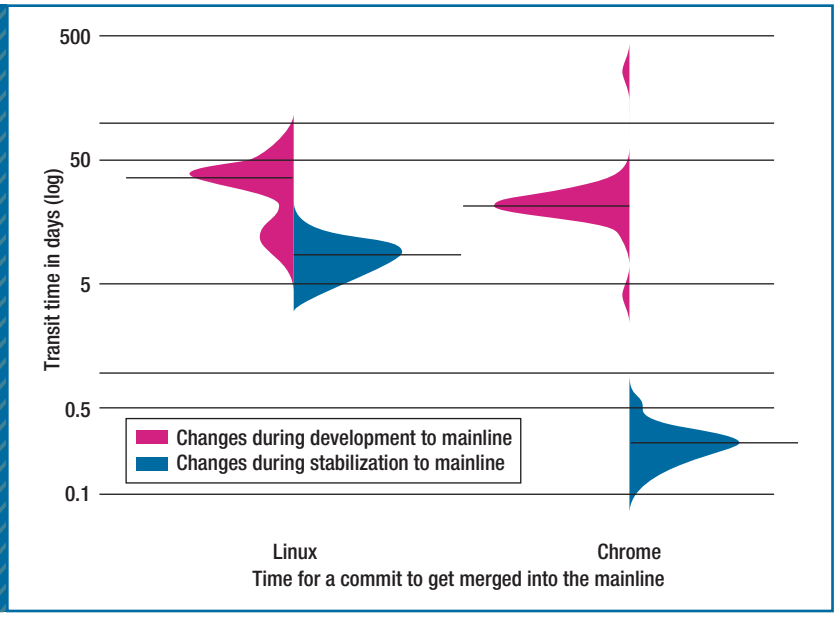


FIGURE 7. The transit time for the commits to the stabilization branch and release. Changes made during stabilization (for example, fixes to regressions) were integrated and released much more quickly than development changes.

developers who didn't remain to fix the bugs in their small code contribution. Instead, a small group of integrators (see Figure 5) handled integration and bug fixes of regressions during stabilization.

An alternative explanation, and a threat to our study's validity, is that integrators were working in other areas of the file and weren't modifying code lines related to the changes made during development. Although a fine-grained, line-level analysis is left to future research, it's surprising that a different developer modified most files that needed modification during stabilization.

For Linux, the rework done by other developers fluctuated dramatically from 214 to 667 lines and didn't show a clear trend. However, the number of developers whose files didn't need rework clearly increased, with an adjusted $R^2 = 0.97$ and $p << 0.001$. This trend likely showed a maturing in the selection of code from external contributors. For Chrome, all three categories increased, indicating an increase in the number of developers contributing to Chrome but obscuring other patterns.

Although release engineers modified a large number of files, few changes were reverted during stabilization. In the median case, there were 104 reverts per stabilization period for Linux and 3 for Chrome. This accounted for only 2.3 percent of the total stabilization commits for Linux and less than 1 percent for Chrome. For Linux, 55 percent of reverts were during stabilization; for Chrome, 76 percent were during development.

How Quickly Do You Release Changes?

The longer a change takes to transit from development through stabilization to release, the longer users will be waiting for bug fixes and features. In highly competitive environments, small differences in release date can be the difference between success and failure. We find that even with "fixed" rapid-release dates, slips in the release schedule still occur.

We wanted to understand how quickly bugs are fixed during stabilization and how quickly new development is incorporated into Linux and Chrome. *Transit time* is the number of days for a change to be integrated in the stabilization branch or included in a final release. Previous research measured the transit time for a change to be released but ignored the different purposes of changes and found large variations in transit times (three to six months).[10] By differentiating between change types, we found that most of the variation can be explained by whether the change was a fix made during stabilization or a change made during normal development.

Figure 7 shows that the median time for stabilization changes (fixes to regressions) to be included in a stabilization branch was only eight days for Linux and less than one day for Chrome. In contrast, the time for development changes to reach the stabilization branch was 35 days for Linux and 21 days for Chrome. A Wilcoxon test showed that these differences were statistically significant at $p << 0.001$.

The transit time for a stabilization change to be released was 47 days for Linux and 78 days for Chrome, whereas a development change took 97 days for Linux and 109 days for Chrome. Although Chrome started release stabilization every six weeks and produced a new release every six weeks, changes to Chrome took longer to reach users than Linux changes. This is because Chrome

stabilized two releases at a time, whereas Linux stabilized only one.

Do You Rush Changes into a Release to Avoid Waiting for the Next One?

Nobody wants to wait for the next release, especially if there's only one release per year. As a release date approaches, developers feel pressured to release features that aren't yet stable and well integrated. This pressure increases with long release cycles because developers might rush changes into a release to avoid waiting for the next one. Chrome switched to a shorter release cycle to avoid pressuring developers into rushing unstable code into a release.

We wanted to empirically test whether developers rushed changes in right before release stabilization and feature freeze. We defined the churn rate as the number of lines changed per day and defined the rush period as two weeks before release stabilization. The rush period corresponds to the Linux merge window and is one-third of the Chrome development period. The normal development period is the period between releases before the rush period begins. On Linux, this is the two months before the merge window opens; on Chrome, it's the development cycle's first four weeks. Because we were interested in development and not integration, we excluded all merge commits. We also used the author date instead of the committer date so that "cherry-picked" changes would be counted during development, not when they were picked. We hypothesized that the churn rate in the rush period would be higher than the churn rate during normal development.

To test this hypothesis, we used the nonparametric Wilcoxon test to compare the two distributions' churn rate. We found a statistically significant difference in the churn rate between normal development and the two weeks before stabilization (p = 0.007 for Linux and p = 0.0008 for Chrome). Figure 8 shows that, in the median case, Linux developers changed 5 KLOC per day for the normal period and 6 KLOC per day for the rush period. The corresponding values for Chrome were 14 KLOC and 17 KLOC. These differences represented an increase in median daily churn during the rush period of 20 percent for Linux and 21 percent for Chrome. Despite a rapid release cycle, some rush still occurred before release stabilization. It would appear that some degree of rush is unavoidable.

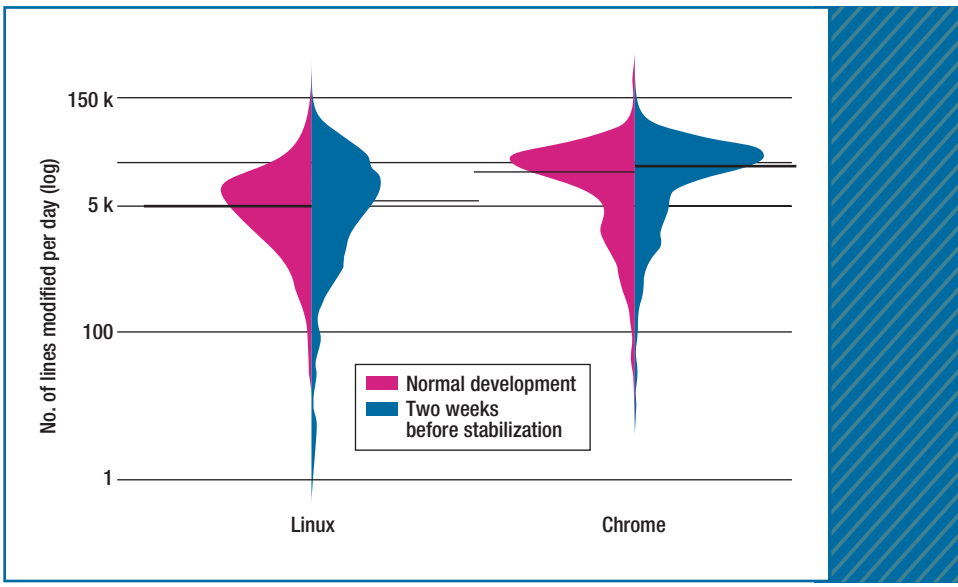


FIGURE 8. The distribution of daily churn during normal development and two weeks before stabilization. The daily churn rate increased by approximately 20 percent during the rush period before release stabilization.

Crosscutting the time and effort measures we examined are the three factors that Facebook's lead release engineer Chuck Rossi considers when creating a release: the schedule, the quality, and the feature set.[4] All three can't be optimized at the same time, so Rossi sacrifices the feature set but releases stable features on schedule and drops any feature that would reduce quality.

Quality is also paramount to Linus Torvalds, whose main jobs are integration and release stabilization. He ranks first in the number of integration merges and 52nd in the number of changes made to Linux. He says,

> *I'm not claiming this [change ...] is really any better/worse than the current behaviour from a theoretical standpoint, but at least the current behaviour is _tested_, which makes it better in practice. So if we want to change this, I think we want to change it to something that is _obviously_ better.*[11]

Likewise, in the article "Release Early, Release Often," Anthony Laforge, who introduced rapid release to Chrome development, states that

> *While pace is important to us, we are all committed to maintaining high quality releases—if a feature is not ready, it will not ship in a stable release.*[12]

So, we conclude that Chrome and Linux value quality over schedule and schedule over features.

References

1. E.S. Raymond, *The Cathedral and the Bazaar*, O'Reilly, 1999.
2. F. Khomh et al., "Do Faster Releases Improve Software Quality? An Empirical Case Study of Mozilla Firefox," *Proc. 9th IEEE Working Conf. Mining Software Repositories* (MSR 12), 2012, pp. 179–188.
3. J. Micco, "Tools for Continuous Integration at Google Scale," Google TechTalk, Google, 2012; www.youtube.com/watch?v=KH2_sB1A6lA.
4. C. Rossi, "Native Mobile App Releases," 2014; www.youtube.com/watch?v=Nffzkkdq7GM.
5. A. Laforge, "Chrome Release Cycle," 2011; bit.ly/1qz4ATj.
6. "How the Development Process Works," Linux; www.kernel.org/doc/Documentation/development-process/2.Process.
7. M.T. Rahman and P.C. Rigby, "Contrasting Development and Release Stabilization Work on the Linux Kernel," *Proc. 2014 Int'l Workshop Release Eng.*, 2014.
8. A. Mockus, R.T. Fielding, and J. Herbsleb, "Two Case Studies of Open Source Software Development: Apache and Mozilla," *ACM Trans. Software Eng. and Methodology*, vol. 11, no. 3, 2002, pp. 309–346.
9. "Release Engineering," Mozilla, 2014; https://wiki.mozilla.org/ReleaseEngineering#Team.
10. Y. Jiang, B. Adams, and D.M. German, "Will My Patch Make It? And How Fast? Case Study on the Linux Kernel," *Proc. 10th IEEE Working Conf. Mining Software Repositories* (MSR 13), 2013, pp. 101–110.
11. L. Torvalds, "Re: IRQF_DISABLED Problem," 2007; https://lkml.org/lkml/2007/7/26/442.
12. A. Laforge, "Release Early, Release Often," *The Chromium Blog*, 22 July 2010; http://blog.chromium.org/2010/07/release-early-release-often.html.

ABOUT THE AUTHORS

MD TAJMILUR RAHMAN is a graduate student in software engineering at Concordia University in Montreal. He's studying release-engineering practices, including feature toggles and branching strategies. His research interests include identifying and mitigating the events that disrupt developers and lead to software failures. Rahman received a BSc in computer science and engineering from Northern University Bangladesh. Contact him at mdt_rahm@encs.concordia.ca.

PETER C. RIGBY is an assistant professor of software engineering at Concordia University in Montreal. His research interests focus on understanding how developers collaborate to produce successful software systems. Rigby received a PhD in computer science from the University of Victoria. Contact him at peter.rigby@concordia.ca.

FOCUS: RELEASE ENGINEERING

Rapid Releases and Patch Backouts

A Software Analytics Approach

Rodrigo Souza and Christina Chavez, Federal University of Bahia

Roberto A. Bittencourt, State University of Feira de Santana

// To investigate the results of Mozilla's adoption of rapid releases, researchers analyzed Firefox commits and bug reports and talked to Firefox's developers. The results show that developers are backing out broken patches earlier, rendering the release process more stable. //

RELEASE ENGINEERING deals with decisions that impact the daily lives of developers, testers, and users and thus contribute to a product's success. Although gut feeling is important in such decisions, it's increasingly important to leverage existing data, such as bug reports, source code changes, code reviews, and test results, both to support decisions and to help evaluate current practices. The exploration of software engineering data to obtain insightful information is called *software analytics*.[1]

In 2011, the Mozilla Foundation fundamentally changed its release process, moving from traditional 12- to 18-month releases to rapid, six-week releases. The motivation was the need to deliver new features earlier to users, keeping pace with the evolution of Web standards, the competition among Web browsers, and the emergence of mobile platforms.

Researchers have used software analytics to study the impact of Mozilla's adoption of rapid releases (see the sidebar). Those studies focused on changes from the viewpoint of users, plug-in developers, and quality engineers. Here, we focus on how rapid releases affect code integration, which is essential for the timely release of new versions.

In particular, we analyze how the *backout* rate evolved during Mozilla's process change. A backout reverts a patch that was committed to a source code repository, either because it broke the build or, generally, because some problem was found in the patch. Backout implies rework because it requires writing, reviewing, and testing a new patch. A high backout rate indicates an unstable process.

Code Integration at Mozilla

Over the last five years, development at Mozilla in general, and Firefox in particular, has intensely applied code review and automated testing at multiple levels, such as unit testing and user interface testing. This process has been supported by tools such as Bugzilla, a bug-tracking system, and Mercurial, a distributed version control system. Here we describe the process before 2011 and the changes that occurred after. Because we analyze only the period between 2009 and 2013, we ignore specifics of the process before 2009 and after 2013.

Before 2011: Traditional Releases

Before March 2011, Firefox development followed a traditional release schedule. Features for the upcoming version were developed along with bug fixes and minor updates for the

0740-7459/15/$31.00 © 2015 IEEE

PREVIOUS STUDIES OF RAPID RELEASES AT MOZILLA

Researchers have been studying the impact of Mozilla's move to rapid releases under multiple perspectives. Although the advantages of releasing features earlier are clear, Christian Plewnia and his colleagues showed that, in the first years of the change, Firefox's reputation was harmed.[1] Some reasons include users being prompted to update the software more often and plug-in developers fearing that new releases would break the API. Since then, Firefox has regained its reputation by implementing silent updates and introducing extended support releases.

Regarding the impact on the development itself, Mika Mäntylä and his colleagues showed that Mozilla had to hire more testers and narrow the testing's scope because rapid releases didn't leave enough time to run all manual tests at every release.[2] This approach seems to be paying off. The number of postrelease bugs hasn't changed significantly after Mozilla moved to rapid releases, as Foutse Khomh and his colleagues showed.[3]

References

1. C. Plewnia, A. Dyck, and H. Lichter, "On the Influence of Release Engineering on Software Reputation," presented at 2nd Int'l Workshop Release Eng., 2014; http://releng.polymtl.ca/RELENG2014/html/proceedings/releng2014_submission__3.pdf.
2. M. Mäntylä et al., "On Rapid Releases and Software Testing," *Proc. 29th IEEE Int'l Conf. Software Maintenance* (ICSM 13), 2013, pp. 20–29.
3. F. Khomh et al., "Do Faster Releases Improve Software Quality? An Empirical Case Study of Mozilla Firefox," *Proc. 9th IEEE Working Conf. Mining Software Repositories* (MSR 12), 2012, pp. 179–188.

current stable release. Major features would be delivered to users only with the release of a major version, which occurred when planned features were implemented and tested. In practice, a new major version took from 12 to 18 months to be released.[2]

Figure 1 summarizes bug fixing at Mozilla at that time. A developer first proposed a source code change as a patch on Bugzilla. That developer then requested a code review from another developer, who approved or rejected the patch. In the latter case, a new patch was written and reviewed. Once the patch was approved, the developer committed it to the code repository.

All developers with commit access committed to and pulled changes from the Mozilla central repository, often abbreviated m-c. Nightly builds were created from m-c so that developers, testers, and other stakeholders could test the most recent changes. Automated tests ran during the build process.

The central repository had to be fairly stable because it was the starting point for developing new features and bug fixes. If the code in m-c failed to compile or broke major features, it prevented testing of new changes. In this case, the developer had to back out the offending commit to stabilize the repository or even fix it right away with another commit. In some cases, the repository was closed to prevent further changes while stabilization occurred.

To prevent m-c from breaking frequently, developers could, besides running tests in their development machines, submit their patches to the Try server before committing them. The Try server checked out a copy of m-c, applied the patch, built the code, and ran automated tests. Because building all the platforms and running all the tests might take hours, developers could choose to build a subset of the platforms and run a subset of the tests.

When developers were confident about their patches, they committed them to m-c. They had to wait for the next build cycle and watch the build to ensure the changes didn't break the build or cause test failures. If a problem happened, they had to back out the bug. So, developers were recommended to commit only if they were available for the next four hours.[3]

Once the change was in m-c and tests passed, the corresponding bug report was updated with the status Resolved and resolution Fixed. If further tests detected a problem, the commit was backed out, and the bug report status changed to Reopened so that it could be resolved again.

2011–2013: Rapid Releases

In March 2011, Mozilla started the six-week release cycles, beginning with the development of Firefox 5. In this cycle, though, Mozilla was still stabilizing the process. Only in June did it create integration repositories, such as Mozilla inbound (m-i).

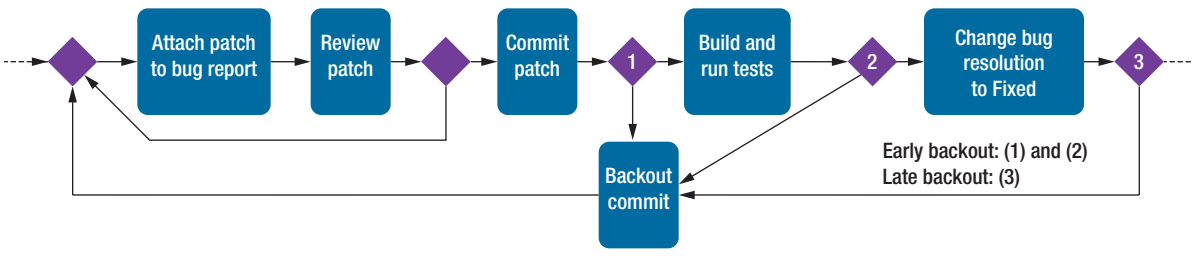


FIGURE 1. Mozilla's bug-fixing process between 2009 and 2013.

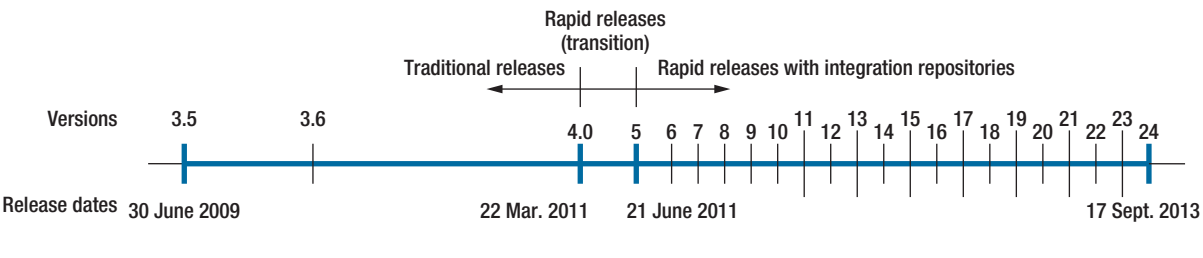


FIGURE 2. Firefox's release history for the periods being studied. The development of version 5 was isolated in the transition period because it was atypical: the release cycle was longer and integration repositories didn't exist then.

Thereafter, patches were committed to m-i, tested on m-i, and then merged once a day with m-c. Patches that broke m-i were backed out before merging.

Instead of each developer watching the build and backing out his or her own commits, build engineers took turns as *sheriffs* who performed this job. Breaking the build on m-i was less of a problem because sheriffs backed out troublesome patches before merging the code into m-c, making it more stable. Sheriffs also changed the bug report status to Resolved, with resolution Fixed, after they tested and merged the patches.

Software Analytics Approach

To investigate how the backout rate changed when Firefox transitioned to rapid releases, we collected, transformed, and analyzed publicly available data produced by Mozilla engineers.

The Data

We used two primary information sources: commit logs and bug reports. We extracted the logs from m-c using the command `hg log`. A Mozilla engineer made the bug reports available as an SQL database dump.

We also obtained release dates from Mozilla's wiki. We analyzed the development of versions 3.6 and 4.0, both developed under traditional release cycles, and versions 5 through 27, developed under six-week cycles. The development of versions 3.6 to 27 amounted to more than four years of data (see Figure 2).

We split the data into three periods: traditional releases, transitional rapid releases, and rapid releases with integration repositories. The development of version 5 was isolated in the transition period because it was atypical: the release cycle was longer, and integration repositories didn't exist then. We mapped each bug report to a release according to the date of its first bug fix commit.

Mapping Commits to Bug Reports

We classified commits as bug fixes or backouts and mapped them to the bugs they fixed or reverted. To this end, we relied on conventions developers use when writing commit messages.

Bug fixes. Bug fix commits start with the word "bug" followed by a five- to six-digit number uniquely identifying the bug. An example is, "Bug 939080 - Allow support-files in manifests to exist in parent paths; r=ted."

TABLE 1

Metrics per release model.

Metric	Release model		
	Traditional	Transitional rapid	Rapid with integration
No. of bug fixes	11,220	1,893	30,085
No. of days	631	90	892
Avg. no. of bug fixes per day	17.8	21.1	33.7
No. of committers*	45.5	64.6	92.7
No. of fixes backed out	702	173	2,831
Fixes backed out (%)	6.3	9.1	9.4
Avg. no. of bugs backed out per day	1.1	1.9	3.2
Early backout rate (%)	3.5	5.1	8.3
Late backout rate (%)	3.1	4.9	1.5
Fixes backed out early (%)†	56.7	55.5	87.7
Median time-to-backout (hrs.)	5.7	12.6	4.2

** Each month, we counted the number of developers with at least five commits; we then averaged this number across the months in each period.*

† The proportion of backed-out bug fixes that were backed out early.

Backouts. Backout commits contain the expression "back out" or a variation, such as "backout," "backs out," "backed out," or "backing out," followed by a 7- to 12-digit hexadecimal number referring to the bug fix commit being backed out, or the number of the bug whose fix it backs out, or both. An example is, "Back out 7273dbeaeb88 (bug 157846) for mochitest and reftest bustage."

Early and Late Backouts

We classified a backout as early if it occurred before the resolution of its bug report changed to Fixed. We classified the backout as late if it occurred after that. In practical terms, an early backout occurs when a commit breaks its first build, usually because it either prevented the code from compiling or made automated tests fail. If a problem is discovered only afterward, a late backout is performed. A bug fix can even be backed out more than once, both early and late.

Data Analysis

After collecting the data, mapping commits to bug reports, and classifying backouts, we determined whether each bug was ever backed out and, if so, whether the backout was early or late. We computed the backout rate as the ratio of the number of bug reports associated with at least one backout commit to the number of bugs associated with at least one bug fix commit.

First, we plotted monthly backout rates. Then, we computed backout-related metrics for the three periods. We applied statistical tests (such as Fisher's exact test and the Wilcoxon signed-rank test) to evaluate whether the differences were statistically significant. Finally, we contacted Firefox engineers, using the firefox-dev mailing list. We reported our numbers and asked them to explain the results according to their experience.

The Results

Here we report on the evolution of backout rate and other metrics, explain why they changed over time, and analyze how they affected Firefox developers and users.

The Numbers

Table 1 shows the metrics for the three periods. First, the number of bug fixes per day almost doubled under rapid releases. This increase is highly correlated with a growth in the number of regular committers. So, we can infer that the developer workload didn't change significantly over the period. As a Mozilla engineer stated,

A developer can only do so much work; growth is mostly adding developers nowadays, not the individual doing more.

The numbers also show that backout was a relevant problem. Under rapid releases, 9.4 percent of all bugs that were fixed eventually got backed out, an average of 3.2 bugs daily.

Figure 3 shows that the overall backout rate increased under rapid releases. A few Mozilla engineers pointed out that the backout rate might have been underestimated under traditional releases because the backout culture became more prevalent after the introduction of sheriff-managed integration repositories. Before that, developers usually fixed a broken commit by recommitting, without explicitly backing out the first commit:

Where bugs might have had broken patches land and gotten fixed in-tree, our current process and tree sheriffs will back-out obvious failures until the bugs get fixed before landing.

[With inbound and sheriffs,] we do not end up fixing the issues with follow-up after follow-up fix but rather have them backed out right away.

To better understand the backouts' impact, we broke them down into early and late backouts. Figure 4 shows that, although the early-backout rate grew, the late-backout rate dropped significantly after the introduction of integration repositories.

Table 1 reinforces that a shift occurred toward earlier problem detection. Among all backouts, the proportion of early backouts increased from 57 to 88 percent. The time-to-backout (the time for an inappropriate bug fix to be reverted) also dropped after the adoption of rapid releases.

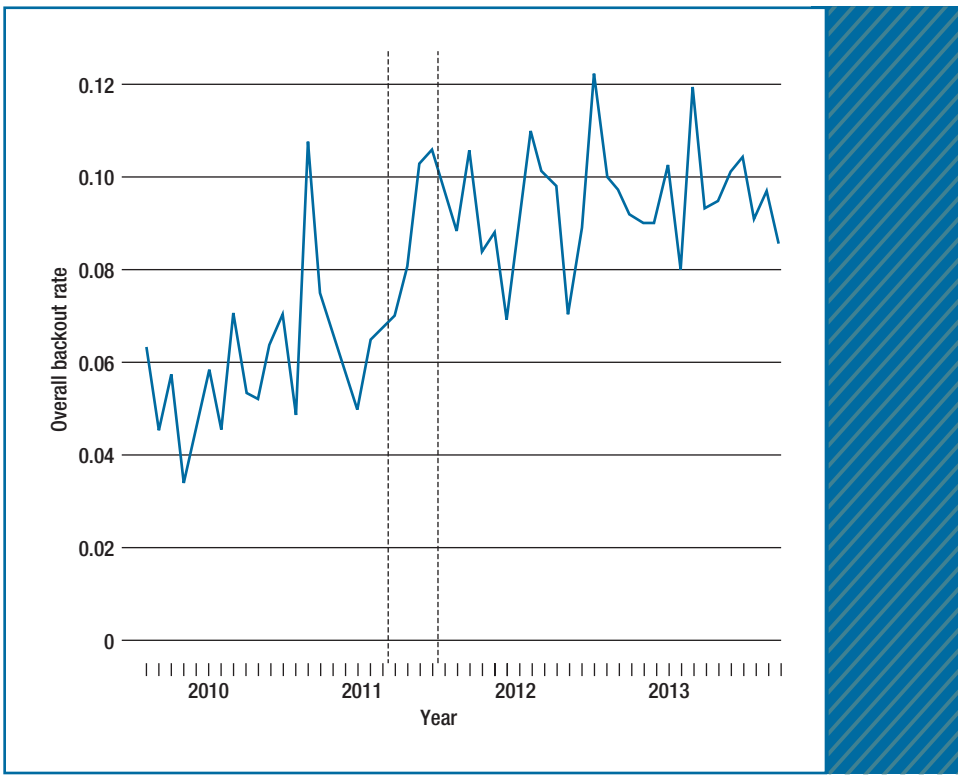


FIGURE 3. The overall backout rate over time. The dashed vertical lines represent important events. The left one is the start of the first rapid-release cycle; the right one is the introduction of integration repositories. The overall backout rate increased under rapid releases.

What Does It All Mean?

What do the backout trends reveal about changes in Firefox's process and context? Eight Mozilla engineers offered explanations.

A larger code base and more products. Some engineers explained the increase in the overall backout rate by suggesting that because the code base grew over time, code conflicts became more likely. The number of supported platforms also increased because Firefox must support both new platforms, such as Windows 8, and older ones, such as Windows XP. Also, new products emerged, such as Firefox for Android and Firefox OS, that share code with the desktop Web browser. As one engineer explained,

We have a lot more stuff that can break, on more platforms, as well as more tests—these days we don't have everyone working on just Firefox. Code landing for B2G [the Firefox OS] can break Fennec [Firefox for Android], for example, and B2G devs don't build and test on Fennec locally. Those kinds of changes will be caught and backed out when they hit the trees [code repositories], not found beforehand.

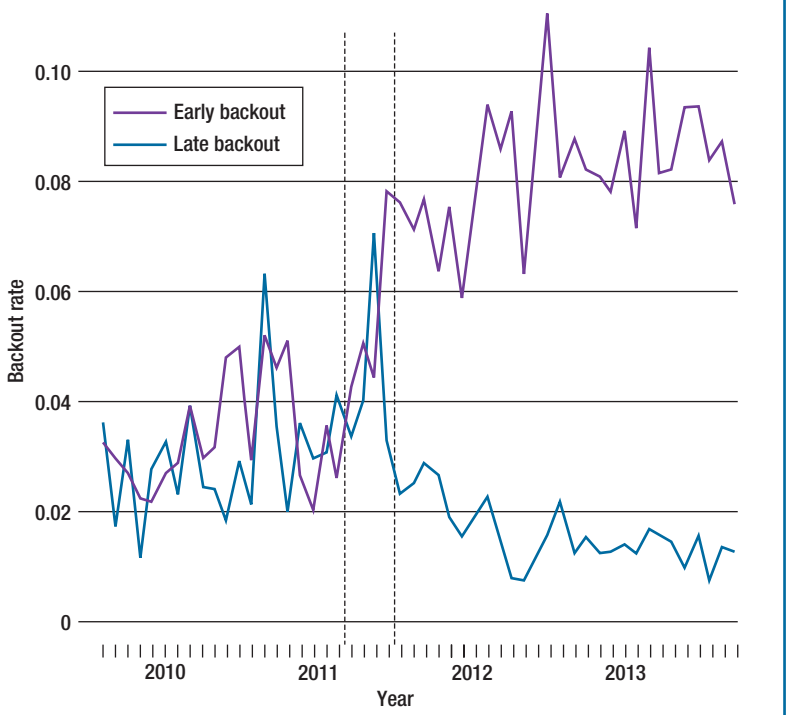


FIGURE 4. The early-backout and late-backout rates over time. The dashed vertical lines represent important events. The left one is the start of the first rapid-release cycle; the right one is the introduction of integration repositories. Although the early-backout rate grew, the late-backout rate dropped significantly after the introduction of integration repositories.

The evolution of testing tools. The increasing early-backout rate and decreasing late-backout rate were due partly to the evolution of the automated testing toolset. According to Mozilla engineers, the emergence of better testing tools promoted earlier detection of problems and improved even detection of problems that would have otherwise gone unnoticed, such as hard-to-detect memory leaks:

> *Our automated testing has improved considerably since [release] 3.5. A number of memory-leak-finding tools have been integrated into our test environments that are improving our early catch rate.*

Integration repositories and backout culture. The increasing early-backout rate was also due to the sheriff-managed integration repositories and their effect on how developers test their code. Before 2011, because developers pushed changes directly to m-c, the changes had to be thoroughly tested to avoid breaking the builds or introducing bugs. From 2011 to 2013, developers committed to integration repositories, and the sheriff backed out problematic patches before merging changes to m-c, thus keeping it stable, as we mentioned before. So, developers were encouraged to commit to m-i after having performed less testing. As someone stated in Mozilla's wiki,

> *But breaking it [the integration repository] rarely is ok. … Never breaking the tree means you're running too many tests before landing [committing to the repository].*[4]

Two Mozilla engineers reinforced this view:

> *In the "old days," you were expected to have built, tested, done a Try build, etc. before the patch landed.*
>
> *The backout aggressiveness was even explicitly mentioned when we switched.*

So What?

To understand what the changes in backout rates mean to Firefox developers and users, we first have to understand the impact of early and late backouts.

The impact on developers. Every backout induces rework by requiring development of a new, improved patch. However, in Mozilla's case, the increase of early backouts didn't seem to cause overhead. Instead, it reflected a cultural change toward committing patches before testing them comprehensively, therefore reducing the effort required to test patches. Such change was possible only because broken patches no longer reached m-c. Sheriffs also ensured that patches that break the build were

backed out as soon as possible, reducing the time in which the repository must be closed:

> *I'd say amount of time spent testing patches before landing, and amount of time wasted with trees closed due to bustage [a broken build], were reduced.*

Although all backouts induce rework, late backouts are severer. First, the longer a fix takes to be backed out, the more time developers spend trying to remember the context and set up their environments to create an improved patch. Also, problems that aren't resolved early might end up in a release. So, users might have to wait another release cycle to receive the definitive bug fix. Finally, with integration repositories, inappropriate commits that weren't backed out early ended up in m-c, on which developers based their work. By the time the commit was backed out, many other commits might have depended on it.

So, from the shift toward earlier backouts, we can infer that the sheriff-managed integration branches reduced the effort required to integrate bug fixes.

The impact on users. Although Mozilla's move to rapid releases was a success from the release-engineering perspective, it upset users because of frequent update notifications and broken plug-in compatibility. As the then chair of Mozilla Foundation summarized on his blog post,

> *We focused well on being able to deliver user and developer benefits on a much faster pace. But we didn't focus so effectively on making sure all aspects of the product and ecosystem were ready.*[5]

However, backouts had no effect on users' perception of quality. This is because, after being committed to m-c, all bug fixes went through two other repositories, aurora and beta, where more tests occurred during two release cycles before they were released to the general public. So, only very late backouts affected users, and these were rare under both traditional and rapid releases. As a Mozilla engineer stated,

> *I think our development process gives us a margin of safety to detect regressions well before the code actually reaches the hands of users. As the user base of each repository grows gradually, we have an effective way to detect unexpected problems well in advance.*

As we mentioned before, Mozilla took two concrete measures that helped keep the process stable while letting it move faster: improving automated testing tools and using integration repositories. You can use these two measures to improve any project. However, the overhead incurred in implementing them is more justifiable under rapid releases because in that context it's important to keep the source code stable as often as possible. For example, having

ABOUT THE AUTHORS

RODRIGO SOUZA is a PhD student in the Federal University of Bahia's Department of Computer Science. His research interests include empirical software engineering, release engineering, software evolution, and mining software repositories. Souza received an MSc in computer science from the Federal University of Campina Grande. Contact him at rodrigo@dcc.ufba.br.

CHRISTINA CHAVEZ is a professor in the Federal University of Bahia's Department of Computer Science. Her research interests include software design and evolution, and software engineering education. Chavez received a PhD in computer science from the Pontifical Catholic University of Rio de Janeiro. She's a member of ACM and IEEE. Contact her at flach@ufba.br.

ROBERTO A. BITTENCOURT is an assistant professor of computer engineering at the State University of Feira de Santana. His research interests include software evolution and design, computing education, and computer-supported cooperative work. Bittencourt received a PhD in computer science from the Federal University of Campina Grande. Contact him at roberto@uefs.br.

frequently stable code is important when Mozilla must deliver a "chemspill" release—one that fixes critical security issues and thus should reach users as soon as possible.

Our analysis uncovered previously unknown information about the evolution of early and late backouts in Firefox and therefore helped evaluate the impact of the adoption of rapid releases by Mozilla. In the future, such analysis could be integrated into an analytics tool constantly updated with backout rates and other metrics. Release engineers would have easier access to up-to-date information about the process, letting them evaluate the impact of yet-to-be-made decisions.

Acknowledgments

We thank all the Mozilla engineers who provided feedback on the results.

References

1. D. Zhang et al., "Software Analytics in Practice," *IEEE Software*, vol. 30, no. 5, 2013, pp. 30–37.
2. "Releases," *MozillaWiki*, Mozilla, 2014; https://wiki.mozilla.org/Releases.
3. "Committing Rules and Responsibilities," Mozilla, 2014; https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Committing_Rules_and_Responsibilities.
4. "Tree Rules/Integration," Mozilla, 2014; https://wiki.mozilla.org/Tree_Rules/Integration.
5. M. Baker, "Rapid Release Follow-Up," blog, 3 Oct. 2011; http://blog.lizardwrangler.com/?p=2996.

Vroom: Faster Build Processes for Java

Jonathan Bell, Columbia University

Eric Melski and Mohan Dattatreya, Electric Cloud

Gail E. Kaiser, Columbia University

// To speed up testing, researchers combined two complementary approaches. Unit test virtualization isolates in-memory dependencies among test cases. Virtualized unit test virtualization isolates external dependencies such as files and network ports while long-running tests execute in parallel. //

SLOW SOFTWARE BUILD CYCLES substantially hinder continuous integration during development. They can be an even more significant nuisance for continuous delivery and other release processes. As a complex software system evolves and its compilation and packaging process becomes more complicated, building changes from a process that developers perform frequently on their desktop machines after every small code edit, to one performed nightly on a dedicated build machine, to one that can't even be performed in its entirety overnight. We aim to significantly reduce build time, with a sufficiently general solution applicable to both full ("clean") and incremental builds.

We decided to reduce building time by reducing testing time. To do this, we developed a system that combines two approaches. The first approach, *unit test virtualization*, isolates in-memory dependencies among test cases, which otherwise are isolated inefficiently by restarting the Java Virtual Machine (JVM) before every test. We call our implementation of this approach VMVM (Virtual Machine in the Virtual Machine, pronounced "vroom vroom"). The second approach, *virtualized unit test virtualization*, isolates external dependencies such as files and network ports while long-running tests execute in parallel. We call our implementation of this approach VMVMVM (Virtual Machine in a Virtual Machine on a Virtual Machine—"vroom vroom vroom").

The Dominance of Testing Time

We've found that the testing phase for real-world Java-based build processes often dominates compilation, packaging, and other traditional contributors to the build time. So, we focus on reducing the clock time needed to run test suites. Some of our industry partners report that they've been forced to remove testing from their regular build process as a stopgap solution. For instance, one partner reported that its Java-based build process took about eight hours—long enough to be problematic even for nightly build cycles.

To obtain concrete data on this problem, we measured the compilation, test, and other build phases for 20 popular open-source Java projects. We found the situation could be even worse than in our partner's anecdote: the testing phase took more than four times as long, on average, as the rest of the build (see Figure 1).

Unacceptably long test cycles aren't new. Previous research tried to reduce the time to run test suites.[1] It focused on

- selecting the smallest subset of relevant tests deemed most likely

0740-7459/15/$31.00 © 2015 IEEE

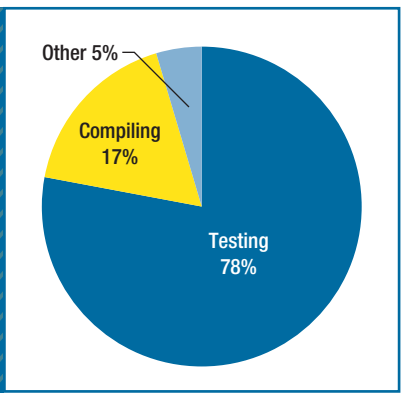


FIGURE 1. How build time is spent. By reducing testing time, we aim to significantly reduce build time, with a sufficiently general solution applicable to both full and incremental builds.

to find the faults for a given change set or
- reordering tests to execute those more likely to fail sooner.

The former approach can find only the defects affected by that change set. The latter approach might find defects sooner but only reduces the total time needed if testing halts after a time-out.

Furthermore, the approach that reduces the number of tests in a suite isn't sound—there's always a risk that some tests are deemed irrelevant when they aren't. (In the general case, it's undecidable whether one test suite is equivalent to another.) So, we seek to reduce testing time while still executing the entire test suite, with no loss of fault-finding ability.

Isolation Inefficiency

We studied the testing process that many Java projects employ, using the popular JUnit framework. JUnit can be used for integration and full system end-to-end tests as well as unit testing. We observed a common yet inefficient practice: each test executed in a fresh process—that is, in its own Java Virtual Machine (JVM).

An important implicit assumption in testing is that the result of test T shouldn't depend on the execution of some previous test T_p. This assumption of independent test cases, a part of the *controlled regression testing assumption* (applicable to other kinds of testing besides regression), is difficult to achieve efficiently.[2] Ideally, it's enforced with pretest setup methods and post-test tear-down methods.

However, for complex software (for example, software that uses black-box third-party APIs), ensuring these methods' correctness can be particularly difficult. The testing code might be buggy or incomplete, just like the application code being tested. Moreover, testers might miss resetting the state of some part of the system under test (and hence cause an unexpected dependency between one test and another). In this case, the results can range from false positives (where tests incorrectly raise an alarm when the code is correct) to, what's worse, false negatives (where tests fail to raise an alarm despite errors in the code).

So in practice, each test often executes in a separate process, which ensures that the tests are isolated (and don't have hidden dependencies), greatly simplifying writing the pretest and post-test methods. This isolation comes at a significant cost. We studied the overhead of executing each test in its own process, relative to the time needed to simply execute each test in the same process, for the 20 Java projects we mentioned earlier. We found it to be astonishingly high: on average, 618 percent (and up to 4,153 percent!).

Completely removing this isolation from the testing process in these applications would reduce application build time a net 56 percent. However, removing test isolation can have disastrous consequences on test suite correctness. In our study, we found 70 test cases that passed in isolation yet failed unexpectedly without it. Even worse, there are reports of test cases that erroneously pass when not isolated (despite a defect in the application under test) and fail only under isolation.

VMVM and VMVMVM

To combat this overhead while maintaining test case isolation, we developed unit test virtualization,[3] which automatically and efficiently isolates the side-effects of unit tests and other tests. The system reinitializes only that part of memory written by some previous test that could be read by the next test (determined by static and dynamic analysis), rather than restarting the entire process to reinitialize the entire in-memory state. This provides the same level of isolation that running each test class in its own JVM would provide. (Multiple test methods in the same test class still can have dependencies, which current versions of JUnit allow.)

We implemented our approach for Java in VMVM. As we show later, when we applied VMVM to the test suites of the 20 projects, it achieved an impressive average net speedup of the entire build time (compared to running each in a separate process). Because all tests executed, no loss of fault-finding ability occurred.

VMVM works well for speeding up test suites when the overhead of restarting the JVM between tests (usually a constant 1 to 2 seconds) constitutes a significant portion of the testing time. However, in cases with only a few test classes that are

very long (for example, 10 seconds each), removing the overhead of restarting the JVM and adding the overhead of our dynamic analysis can slow down testing.

In those latter cases, we take a complementary approach to reducing clock time: we leverage modern multicore hardware to execute multiple test cases in parallel. A simple approach to parallelizing test cases (offered by the most recent versions of Ant and Maven) uses a controller thread to distribute test cases in round-robin manner to several workers executing in parallel on the same machine. (This simply spawns extra processes running in the same directory.) However, of our 20 projects, five had tests fail erroneously with this parallelization, with test cases racing for access to shared resources (for example, files or sockets). To safely execute multiple tests simultaneously, each must have its own virtual file system and network interface.

So, we developed virtualized unit test virtualization, which leverages a distributed architecture. Each worker process executes in a distinct conventional VM (as in VMware or VirtualBox), with its own virtual file system and other system resources. This architecture is effective at simultaneously executing multiple tests that use local files and network resources (for example, binding to a socket and connecting back to that socket). However, it doesn't address test cases that interact with remote servers and databases; such interactions are usually avoided during testing and didn't occur in the test suites we examined. As we mentioned before, we call our implementation for Java VMVMVM.

The result is an integrated two-tier system that reduces the build time for Java projects by reducing testing time in two ways. First, VMVM reduces the time between short test cases that's taken to isolate the test cases. Second, VMVMVM reduces the total time for long test cases by letting them run in parallel (and using VMVM internally so that the same JVM can be used for the sequence of test jobs run in the same worker VM).

> We leverage modern multicore hardware to execute multiple test cases in parallel.

The Problem Scope

To empirically ground our efforts to reduce build time, we asked three main questions:

- Does testing take a significant portion of build time?
- Do developers isolate their test cases?
- If they do, is this isolation sufficient to let tests run in parallel?

To answer these questions, we downloaded the 1,200 largest free and open-source Java projects from the indexing website Ohloh (now Open Hub; www.openhub.net). From those, we selected the projects that executed JUnit tests during their Ant- or Maven-based builds. We tried to build all the 591 projects that use JUnit but found that only about 50 worked out of the box without significant configuration (for example, worked by running a single command such as `ant test` or `mvn test`). From those, we selected the 20 projects we've been talking about. This was a manageable set of projects that ensured a diversity of widely used, recognizable projects (for example, the Apache Tomcat JavaServer Pages server) and smaller projects (for example, JTor, an alpha-quality Tor implementation with a very small contributor base).

Measuring Testing Time

To answer the first question, we executed the entire build process for each application (using its Ant or Maven build script), recorded the time each build step took, and aggregated the time for all the testing (junit) steps and all the compilation (javac) steps. We executed this process 10 times, averaging the results.

Table 1 shows the results, which roughly matched our expectations based on our anecdotal industry evidence. For this study, we ensured that all tests were isolated in their own process; those projects that already performed this isolation are bold in Table 1. Testing took on average 78 percent of the build time.

Isolating Test Cases

To answer the second question, we statically analyzed the test scripts for the 591 projects to determine the percentage of them that executed each test case ("test class" in JUnit terminology) in its own process. Of those projects with the most test cases (over 1,000 test cases; 47 total), 81 percent executed each test in its own process. Overall, 41 percent of all the projects executed each test in its own process.

TABLE 1

The build speedup for 20 popular open source Java projects.*

Project	No. of classes	Test LOC × 1,000	Build time spent testing (%)	Build speedup (%)	
				VMVM†	VMVMVM‡
Apache Commons Codec	46	17.99	91	83	85
Apache Commons Validator	21	17.46	93	31	34
Apache Ivy	119	305.99	95	70	86
Apache Nutch	27	100.91	92	13	16
Apache River	22	365.72	74	41	43
Apache Tomcat	292	5,692.45	99	28	68
betterFORM	127	1,114.14	98	73	90
Bristlecone Performance Test Tools	4	16.52	94	–2	12
btrace	3	14.15	49	23	–20
Closure Compiler	223	467.57	93	63	75
Commons IO	84	29.16	96	52	85
FreeRapid Downloader	7	257.70	43	43	41
gedcom4j	57	18.22	98	57	75
JAXX	6	91.13	48	45	34
Jetty—Java HTTP Servlet Server	6	621.53	64	18	16
JTor	7	15.07	61	64	63
mkgmap	43	58.54	88	59	68
Openfire	12	250.79	32	32	31
Trove for Java	12	45.31	56	60	59
Universal Password Manager	10	5.62	97	95	70
Average	56	475.30	78	47	52

** Bold indicates that, in the default configuration, the build isolated each test by executing it in its own process and ran all the test processes sequentially in the same OS on the same machine (no virtual machines). Otherwise, the default configuration didn't isolate tests but ran them all in the same process.*

† Virtual Machine in the Virtual Machine

‡ Virtual Machine in a Virtual Machine on a Virtual Machine

Test isolation is necessary for many complex software systems. Otherwise, the testers would need to write additional test cases for the pretest setup and post-test tear-down methods, to test the tests. Kıvanç Muşlu and his colleagues pointed out a perfect example of what can happen when tests aren't isolated.[4] They found that a fault that took four years to resolve (Apache Commons CLI-26, 186 and 187) could have been detected immediately (even before users reported it) if the project's test cases had been isolated.

The problem was that several test cases checked the Apache library's behavior under varying configurations, and their setup stored these configurations in a `static` field. However, other tests assumed that the system under test would be clean, in a default configuration. But some of the configuration-modifying tests

happened to be earlier in the test suite and didn't restore the `static` field when finishing. So, the later tests in the test suite that should have caught the defect passed (because the defect occurred in only the default configuration), and the defect went undetected.

In Sai Zhang and his colleagues' sample of popular open-source Java software, 96 tests displayed similar dependencies.[5] Of those dependencies, 61 percent arose from side effects from accesses to `static` fields.

Isolation and Parallelism

Our final motivating study addressed the need to enforce further isolation than process separation provides, when tests run in parallel. Although executing each test in its own process eliminates in-memory dependencies between test cases, other persistent state could cause dependencies among tests. For example, when multiple tests read and write from the same file on disk, the test run's results might depend on their execution order. Even if each test properly cleans up after itself (for example, deleting the file), these tests still can't execute concurrently on the same machine because they would compete for simultaneous access to the same file.

We executed the test suites for each of the 20 Java projects several more times. We isolated each test case in a separate process but ran up to eight tests from each test suite concurrently on the same machine (using the parallelization option available in the most recent versions of Ant and Maven). As we mentioned before, five projects (Apache Ivy, Apache Nutch, Apache Tomcat, mkgmap, and Jetty) had tests fail erroneously when executed concurrently. Moreover, even more failures might have occurred; we didn't explore all possible scheduling combinations in which tests might execute concurrently.

Examining these five projects' source code, we found two sources of dependencies: files and network ports. For example, some tests created temporary directories, wrote files to them, and deleted the directories when the test ended. This caused conflicts when two test cases executed simultaneously. Both tests used the same temporary directory; the test that finished first deleted the directory, causing the second test to unexpectedly fail.

We also saw several cases of conflicting bindings to network ports. In these cases, part of the test setup started a mock server listening to some predefined port and then connected the code under test to that port. The first test to bind to the port succeeded; subsequent tests that executed while that first test was running unexpectedly failed, unable to bind to the port. None of the observed dependencies occurred when the test suites executed serially; all the tests correctly cleaned up the environment state at their conclusion. None of these projects had conflicts on resources external to the machine (for example, remote servers).

> We found two sources of dependencies: files and network ports.

Reducing Testing Time

Our key insight is that we can provide the same level of test case isolation as process separation without paying the high overhead cost of restarting the JVM for each test case. The typical reason engineers ask the build process to restart the JVM before every test case is to eliminate in-memory dependencies between test cases. Our insight relies on the observation that these dependencies are easy to track and manage within a single JVM (without restarting) with much lower overhead and thus shorter testing time. Then, we found that we can speed up testing further by parallelizing—running multiple tests simultaneously—if we can also remove system-level dependencies.

Isolating Object Graphs

The JVM provides a managed memory environment in which code can't construct pointers to arbitrary memory locations. Instead, the memory M accessible to some executing function F is constrained to only that memory reachable from F's object graph, plus any `static` fields. The object graph encompasses the traditional object-oriented view of memory: F can receive several pointers to objects as parameters, those objects can in turn have pointers to other objects, and so on.

It's easy to imagine how to isolate this object graph between test executions. Assume that the test runner (which is instantiating each test) constructs new arguments to pass to each test case and doesn't pass a reference to any of the same objects to multiple tests, as would normally

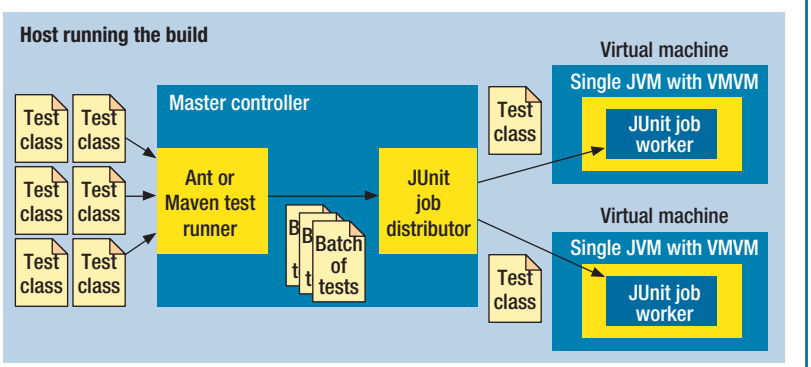


FIGURE 2. The high-level architecture of VMVMVM (Virtual Machine in a Virtual Machine on a Virtual Machine). Our system integrates directly with test execution initiated by Ant or Maven, or via JUnit directly. Each test class execution is intercepted and sent to a master controller that delegates test cases to workers.

be the case. Then (at their creation) no two tests' object graphs will have overlapping nodes. Because the test runner is standardized to the testing framework, ensuring that tests at this level are isolated is easy.

Isolating static Fields

In contrast, Java **static** fields are like global variables: they're directly referenced by their field name and class name (no additional pointers needed). So, we must isolate them.

Our approach to isolating **static** fields is simple and emulates exactly what happens when the JVM restarts. Between each pair of test cases, we reexecute the initializer for every **static** field, effectively eliminating **static** fields as a source of dependencies between tests. VMVM optimizes this basic approach to further reduce the overhead of isolating test cases. It reinitializes only the mutable **static** fields of classes used during prior test executions when they're needed and always ignores fields that are immutable (guaranteed to be unchanged).

VMVM performs offline static analysis and bytecode instrumentation before test execution. This analysis and instrumentation occurs each time the application code or tests change. In this phase, VMVM determines which classes contain no mutable **static** fields and thus won't ever need reinitialization.

VMVM then emulates exactly the process JVM uses internally for initializing a class. It inserts guards (in the bytecode) around every access to the class to check whether the class must be reinitialized and, if so, to reinitialize it. It inserts these guards before every instruction that might create a new instance of a class (the **new** bytecode instruction), access a class's **static** method (the **INVOKESTATIC** bytecode instruction), or access a class's **static** field (the **GETSTATIC** and **PUTSTATIC** instructions). VMVM also intercepts calls to Java's reflection library that would dynamically perform the same operations, adding guards on the fly. In addition, it modifies each class initializer to insert instructions to log its execution. This lets VMVM efficiently determine exactly which classes were used by previous test cases and thus will need to be reinitialized in the next test case that references them.

VMVM performs all these instrumentations on only the application bytecode (not code in the Java core library set). To reinitialize **static** fields belonging to classes in the Java core libraries, we wrote a tool that scans the Java API to identify public-facing methods that set **static** fields. We then verified each result by hand (this process would have to be repeated only for new versions of Java). We found 48 classes with methods that set the value of some **static** field in the Java API. For each of these methods, VMVM provides copy-on-write functionality, logging each internal field's value before changing it and then restoring that value when reinitializing. To provide this support, VMVM prefaces each such method call with a wrapper to record the value in a log and then scans the log at reinitialization (between each pair of test cases) to restore the value.

Running Tests in Parallel

As we mentioned before, our VMVMVM prototype lets us execute test cases in parallel without interference by employing a conventional VM to ensure that each simultaneously executing test has its own file system and virtual network interface (see Figure 2). VMVMVM still relies on a test's manually written pretest and post-test methods to clean up system resources between test classes, so that no two tests are dependent as a result of some shared file. In our study of the 20 projects, we found no test classes that were dependent (when executed sequentially) because of shared system resources. Other researchers have confirmed that such dependencies are uncommon.[5]

To run *N* tests in parallel, we create *N* VMs, with a single daemon running in each one. Each daemon listens for requests from our master controller process, executes the tests submitted by the controller, and returns the results. The controller collects the results, reorders them to appear as if they executed serially, and returns them to the original invoker of the test suite as if they had executed sequentially on the same machine. The daemons use VMVM to provide in-memory isolation between test cases, so they don't start a new JVM for each test case.

For easy integration, we provide a drop-in replacement for the Ant JUnit task, the Maven JUnit target, and a custom JUnit runner. Engineers need only change their build configuration to use our JUnit target (which accepts the exact same arguments as the normal target); test cases are automatically parallelized.

For instance, when using our Ant task, VMVMVM will automatically start a local socket server, spin up worker processes, distribute the test requests, and return the results (in serial order) to the Ant task. Existing test listeners and custom test runners continue to work normally.

Evaluation

We evaluated how our approaches reduced the 20 projects' build time. For each application, we first ran the entire test suite with each test case isolated in its own process (the baseline configuration). Then, we ran the suite with all tests executing in the same process, but using VMVM to provide isolation. Finally, we ran the suite distributed across three workers, each one running all its tests in the same process, again with VMVM providing the isolation. We performed this entire process 10 times, averaging the results.

We performed this study on our commodity server running Ubuntu 12.04.1 LTS (Long Term Support) and Java 1.7.025 with a four-core 2.66-GHz Xeon processor and 32 Gbytes of RAM. Each worker ran in its own VMWare Workstation 10 VM, running Ubuntu 12.04.1 LTS and allocated 2 Gbytes of RAM and two cores.

Table 1 shows the results. All speedups are relative to the length of a build that isolated each test by executing it in its own process and then ran all the test processes sequentially in the same OS on the same machine (no VMs). If this was a project's default configuration, the table shows it in bold; otherwise, the default configuration didn't isolate tests but ran them all in the same process.

The average speedups provided by both solutions (VMVM alone and VMVMVM parallelized in multiple VMs) were comparable. Build time decreased by 47 percent when we used VMVM to isolate test cases and by 52 percent when we added VMVMVM.

We were interested most in the cases in which one approach significantly eclipsed the other. For example, for Apache Tomcat, VMVM sped up the overall build by only 28 percent, whereas VMVMVM sped it up by 68 percent. For btrace, VMVM sped up the overall build by 23 percent, whereas VMVMVM—with VMVM included—slowed it down by 20 percent. Tomcat had almost 300 test classes, with a fairly even distribution of test lengths, so parallelization was quite effective. On the other hand, btrace had only three test classes, taking 1,410 ms, 36 ms, and 23 ms, respectively.

For btrace, parallelization provided no significant benefit because a single test class dominated the testing time. The communication overhead of distributing the tests to the workers showed through, causing VMVMVM to provide a slowdown compared to VMVM alone. In the other applications in which VMVMVM didn't perform as well as VMVM, the overall number of test classes was nearly the same as the number of workers (three), and one or two of the tests dominated the others in execution time. In such cases, parallelizing test classes wasn't effective; using only VMVM increased speedup.

> In projects with a diverse range of test classes, VMVMVM greatly reduced the time to run a complete build.

Our study shows that in projects with a diverse range of test classes, VMVMVM greatly reduced the time to run a complete build. On popular open source software, such as Apache Tomcat, this reduction was huge. We've released a stand-alone version of VMVM under an MIT license via GitHub (https://github.com/Programming-Systems-Lab/vmvm). We're working with our

industrial partners to release a full version of VMVMVM. We hope our efforts to reduce Java build times can help relieve release engineers from long-running builds.

Acknowledgments

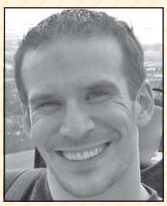
Jonathan Bell and Gail Kaiser are members of Columbia University's Programming Systems Laboratory, which is funded partly by US National Science Foundation awards CCF-1302269, CCF-1161079, and CNS-0905246 and US National Institutes of Health grant U54 CA121852.

References

1. S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, 2012, pp. 67–120.
2. G. Rothermel and M.J. Harrold, "Analyzing Regression Test Selection Techniques," *IEEE Trans. Software Eng.*, vol. 22, no. 8, 1996, pp. 529–441.
3. J. Bell and G. Kaiser, "Unit Test Virtualization with VMVM," *Proc. 36th Int'l Conf. Software Eng.* (ICSE 14), 2014, pp. 550–561.
4. K. Muşlu, B. Soran, and J. Wuttke, "Finding Bugs by Isolating Unit Tests," *Proc. 19th ACM SIGSOFT Symp. and 13th European Conf. Foundations of Software Eng.* (ESEC/FSE 11), 2011, pp. 496–499.
5. S. Zhang et al., "Empirically Revisiting the Test Independence Assumption," *Proc. 2014 Int'l Symp. Software Testing and Analysis* (ISSTA 14), 2014, pp. 384–396.

ABOUT THE AUTHORS

JONATHAN BELL is a PhD student in software engineering at Columbia University. His research interests include software testing, program analysis, and fault reproduction. Bell received an M Phil in computer science from Columbia University. He's a member of the IEEE Computer Society. Contact him at jbell@cs.columbia.edu.



ERIC MELSKI is the chief architect at Electric Cloud and has been developing build optimization software there for more than 12 years. His research interests include distributed systems, high-performance computing, parallel programming, and kernel development. Melski received a BS in computer science from the University of Wisconsin. Contact him at eric@electric-cloud.com.

MOHAN DATTATREYA is the senior director of engineering at Electric Cloud. His research interests include software-defined networks, application acceleration, and distributed-systems performance engineering. Dattatreya received an MS in computer science from Stanford University. Contact him at mohan@electric-cloud.com.

GAIL E. KAISER is a professor of computer science at Columbia University. Her research interests include software reliability and robustness, information management, social software engineering, and software development environments and tools. Kaiser received a PhD in computer science from Carnegie Mellon University. She was a founding associate editor of *ACM Transactions on Software Engineering and Methodology* and has been an editorial board member of *IEEE Internet Computing*. She's a senior member of IEEE. Contact her at kaiser@cs.columbia.edu.

FEATURE: DYNAMIC RECONFIGURATION

Creating Self-Adapting Mobile Systems with Dynamic Software Product Lines

Nadia Gámez, Lidia Fuentes, and José M. Troya,
University of Malaga

// Mobile systems must cope with continuous context changes, making them an ideal fit with dynamic software product lines, which enable product adaptation at runtime. In this DSPL-based process, devices upload only a small reconfiguration plan rather than the entire variability model, and mobile systems manage diversity without disrupting the base model. //

EMERGING MOBILE ecosystems are composed of networked, heterogeneous devices with long lives and resources that are both limited and intermittently accessible. Because they often operate in adverse conditions, these devices must also cope with rapid context changes, yet their reliability, durability, and power-awareness levels must remain high. To manage this balance, mobile systems must be able to self-adapt.

Dynamic software product lines (DSPLs) are a popular paradigm for developing self-adaptive systems. The idea is to model elements as dynamic variation points[1,2] that allow the software to adapt at runtime to changing requirements. DSPL is highly suitable for producing self-adapting mobile systems, which use variability models at runtime to generate successive configurations in response to context changes. Research on DSPL-based self-adaptation is growing, but so far it has addressed only ways to model reconfiguration decisions,[3] not the specific problems that must be overcome to make the approach more practical for mobile systems.

To address that gap, we looked at five main challenges in creating DSPL-based self-adaptation in this domain:

- *Context definition*. The context must consider the requirements of all the actors in a mobile system, such as user, device, and network.
- *Context changes*. Each context change is a trigger for potential reconfiguration. More than one configuration might meet the new context requirements, so any selection mechanism must also consider the quality of service (QoS) for each reconfiguration.
- *Reconfiguration constraints*. Reconfiguration must balance efficiency with the assurance that adaptations will preserve both QoS and desirable system properties.
- *Device heterogeneity*. Generated reconfiguration modules must

0740-7459/15/$31.00 © 2015 IEEE

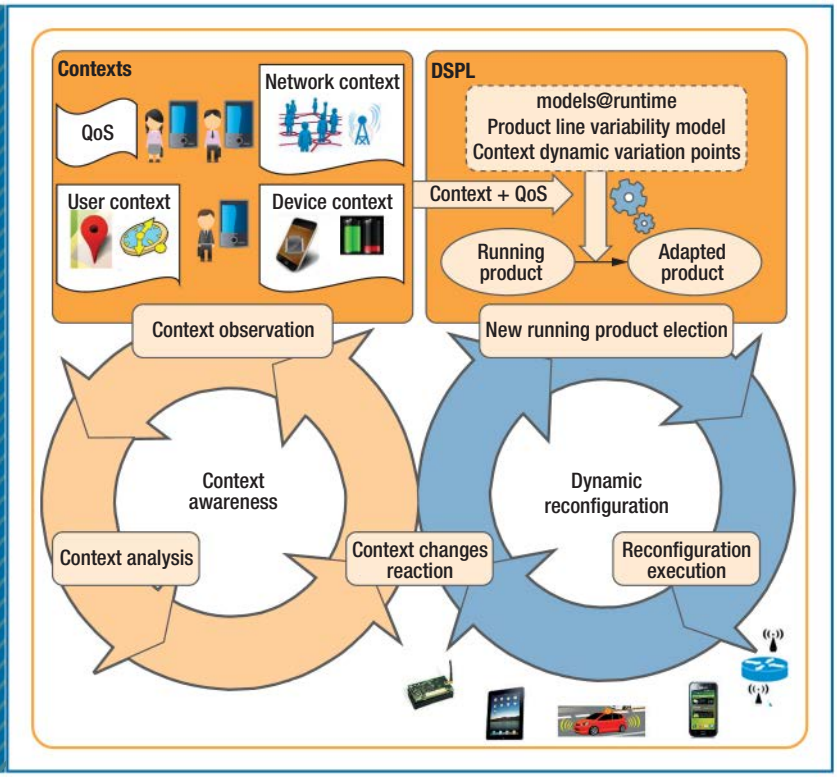


FIGURE 1. A proposed self-adaptation process for mobile systems based on dynamic software product lines (DSPLs). The process has two main cycles. The first cycle observes the context to determine whether a context change that needs the system adaptation has happened. If so, the second cycle will be activated to dynamically reconfigure the system in a way that best meets quality-of-service (QoS) standards and preserves system properties.

adapt to each device's characteristics, and the system must maintain its correct functioning after each device reconfiguration.

- *Language issues*. Nonstandard languages and formalisms are unfamiliar to most mobile application developers. Widespread use of a DSPL-based approach will require a simple standardized language that these developers can recognize.

With these challenges in mind, we created a DSPL-based self-adaptation process that could meet mobile system requirements such as context and device heterogeneity and QoS level. Our process is based on the models@run.time approach, which aims to extend the applicability of models and abstractions to the runtime environment, thereby managing software behavior as it executes.[4] Our lightweight version uses variability models to generate mobile system configurations at runtime. Adaptation is performed at the model level and separate from the base model. Consequently, devices don't need to upload large variability models, only a reconfiguration plan, which can be a tenth of the variability model size. Not all reconfiguration choices will achieve the desired QoS, so reconfiguration selection is based on the desired QoS and system properties.

We implemented our proposed approach in FamiWare, our middleware solution for developing complex mobile computing systems, and applied it to a dynamic route-planning service in a vehicle-to-infrastructure scenario. Our approach met the reconfiguration time constraints to avoid losing position information that might be vital in an emergency situation. We've also verified that our approach efficiently uses memory and battery and performs well with multiple types of smartphones and sensors.[5]

Process Cycles

As Figure 1 shows, our self-adaptation process has two main cycles. The *context-awareness* cycle consists of observing context changes, analyzing new context, and deciding how the system must react to these changes. In DSPL terms, context change starts the process of replacing the product line's current product—the running product—with a new product adapted to the context—the adapted product. In a mobile system, the current product is the current configuration; the adapted product is the new configuration, which is the result of self-adaptation.

If the context change requires self-adaptation, it triggers the start of the *dynamic reconfiguration* cycle. This cycle selects the new running product (adapted product) and executes system reconfiguration. Not all possible configurations meet a desired QoS. For example, the reconfiguration process in a low-energy context could consume more energy than the reconfigured system would save. Therefore, the mecha-

nism that chooses the best configuration from the candidate configurations must trade off reconfiguration and QoS goals.

Context Definition

At any moment, the contexts of all the actors in a mobile ecosystem determine its overall context. For example, system connectivity can depend on the user's location (rural or urban), the device's energy (high or low battery), or the number of devices in the system. Accordingly, the aim of context observation is to determine location or time, energy or resources, and connectivity or number of devices.

Context analysis evaluates the resulting data to ascertain if a context change has occurred. If so, the reaction to context change is self-adaptation, which essentially links the context awareness and reconfiguration cycles.

Dynamic Reconfiguration

In the reconfiguration cycle, the first step in DSPL terms is to choose the adapted product (new configuration) that will replace the running product (old configuration). The variability model, which specifies the points for dynamic modification and the context types, is the basis for generating candidate configurations. New running product election involves choosing the best configuration from these candidates.

To model variability, our approach uses the Object Management Group's proposed Common Variability Language (CVL),[6] which doesn't require any knowledge of complex formalisms and allows variability modeling apart from the base model. However, the same CVL tools can manage both models.

A strong advantage of using CVL is the ability to add variability to base models without having to remodel them. It's also easy to specify models of heterogeneous devices with different reconfiguration needs because CVL allows the expression of variable elements with cardinality and permits element cloning.

The system uses the CVL model to automatically generate the initial configuration and successive adapted configurations and evaluates QoS before selecting a configuration to become the adapted product.

Once the adapted product is selected, the system must implement the specified modifications—the reconfiguration execution stage. Runtime modeling enables adaptation at the model level, so instead of defining heavy model-synchronization engines, mobile application developers simply define a correspondence between the architectural model and the code installed at each moment in each device. They can manage diversity and complexity at the model level, enabling them to modify reconfiguration plans even at runtime, and the correspondence guarantees that the system will enter a valid state after executing a reconfiguration.

Implementation in FamiWare

Figure 2 shows how FamiWare middleware implements our self-adaptation process and CVL's role in modeling system variability and creating the model transformations that support dynamic reconfiguration.

So far, we've implemented FamiWare self-adaptation services for smartphones in Java for the Android platform. However, because our approach is based on DSPL, we can easily add services for other platforms and have already done so for two sensor platforms.[7]

Context Observation and Analysis

To observe context, FamiWare provides services to monitor user, device, and network contexts for both smartphones and sensors. All the services have the same basic structure of reading and publishing context information, and since providers model the services as part of the variability model, service functions can vary according to each system's monitoring needs. Providers can use FamiWare's SPL-based facilities to automatically generate the code to add monitoring services.[7]

The context-awareness service receives events with the monitored data and checks if that information indicates that a context change has occurred. Logic conditions in the variability model serve as the criteria for determining a context switch. Providers can modify these at runtime to incorporate new context conditions on the fly.

A detected context change could require self-adaptation to maintain proper operating conditions or to optimize the system.

At any moment, the contexts of all the actors in a mobile ecosystem determine its overall context.

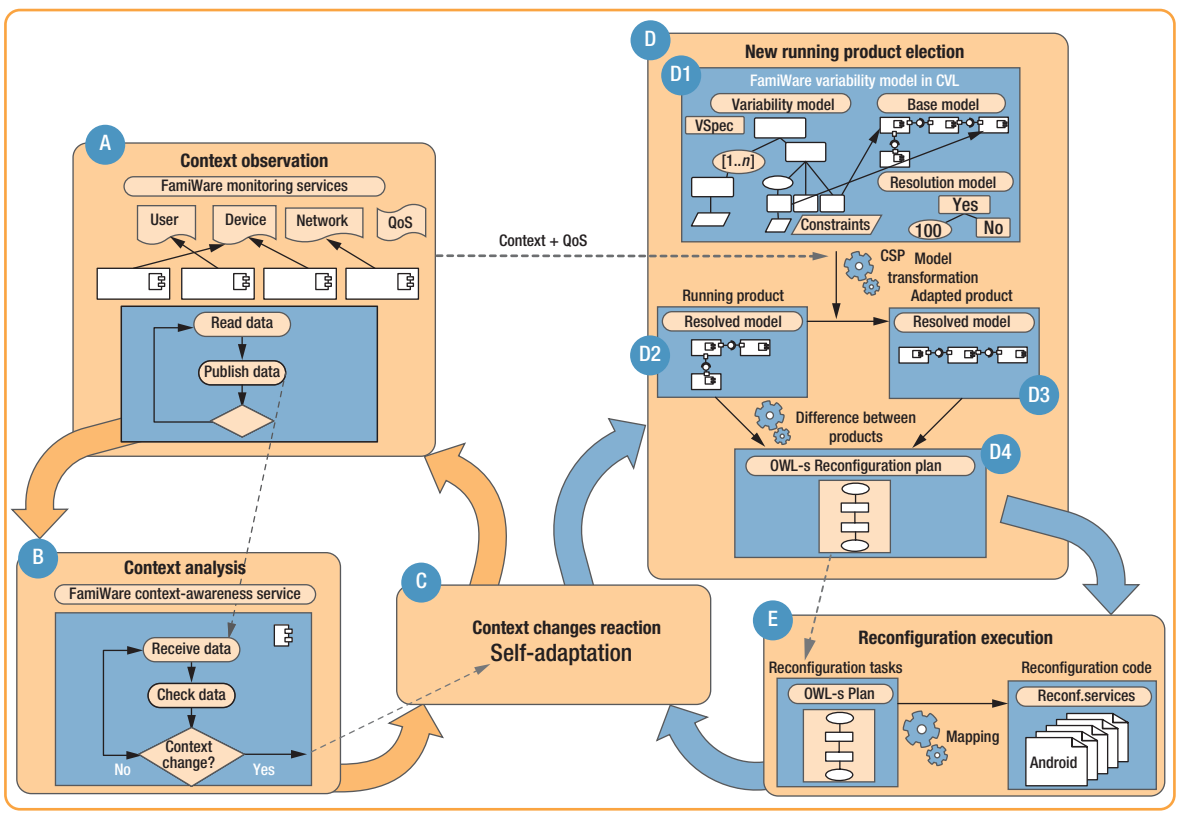


FIGURE 2. Implementation of the self-adaptation process in FamiWare and the Common Variability Language (CVL). FamiWare provides a framework for self-adaptation, while CVL provides mobile system application developers with an intuitive language for modeling variability without having to modify the base model.

Context Adaptation and System Reconfiguration

During design, the variability model generates the initial product, and during self-adaptation at runtime, the model is the basis for generating configurations that meet the new context requirements.

Variability modeling. Figure 2 shows variability modeling with FamiWare. Mobile system application developers use CVL tools to specify both VSpec, FamiWare's variability model, and the binding between VSpec and the base model written in Unified Modeling Language. Resolution models, specified in CVL, select the set of variation points in the variability model that represent a particular product, such as an Android smartphone.

Each device (smartphone or sensor) appears in the model as a CVL variable element with cardinality ([1..*n*]). For each device, FamiWare clones these elements to represent the number of that device type. Each clone can have a different configuration even though it belongs to the same type. This arrangement ensures that reconfiguration accounts for both local and global requirements.

CVL modeling produces variable elements with more detail than is possible in other variability languages. Allowing both fragment substitutions and parameter values means that reconfiguration specifications can be both coarse- and fine-grained. Base-model designers need not be variability modeling experts because they can use CVL tools to express the correspondence between variation points and the base model.

The variability model also defines constraints between context conditions and variation points. Figure 3 shows

three constraints for a location-based service that uses different technologies to acquire users' positions given satellite context, which is the number of visible GPS satellites, the nSat parameter. Technologies might combine GPS, assisted-GPS (AGPS), Global System for Mobile Communications (GSM), and Wi-Fi to obtain a location.

The constraints are essentially the context-change conditions and the required reconfiguration for every context change. For example, if fewer than two satellites are visible, a device can use any location technology except GPS and AGPS. Given this constraint, if a device using a GPS module to obtain user location suddenly can't find any satellites, the system knows that it must modify the device to use the GSM or Wi-Fi modules.

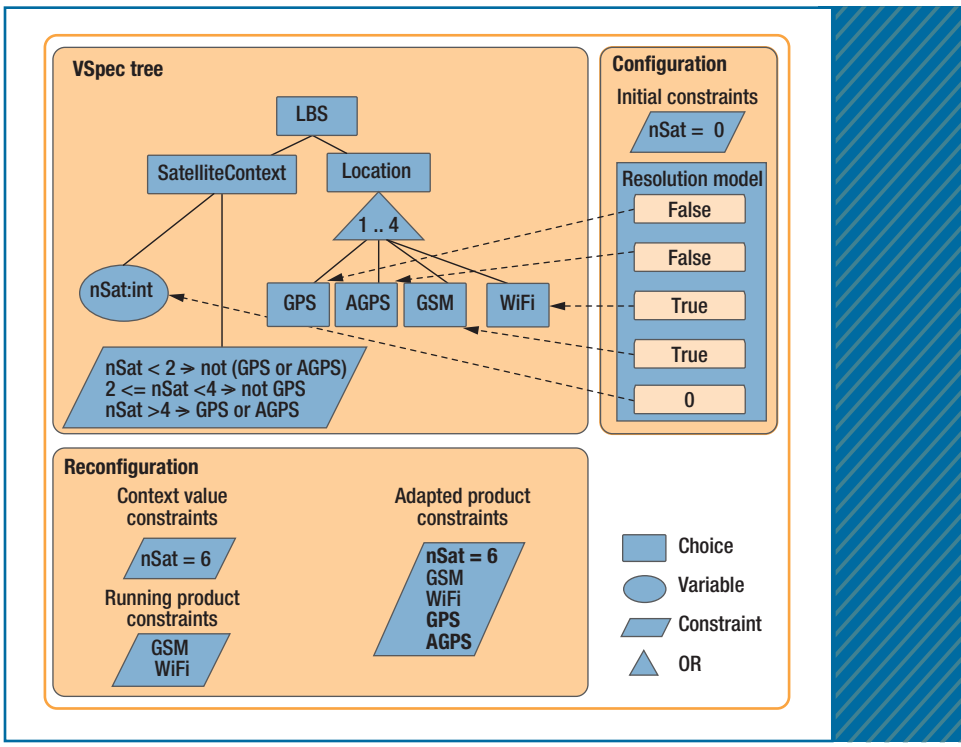


FIGURE 3. Excerpt of a VSpec tree in the CVL variability model for a location-based service that uses a variety of technologies to determine a user's position. Reconfiguration constraints reflect context-change conditions, such as the number of visible GPS satellites.

Initial configuration. Initial context values are initial constraints, such as nSat = 0 in Figure 3. The CVL resolution model reflects these constraints and ensures that the system automatically selects the other variable elements that represent the different system components or services. The CVL resolved model captures the correspondence between the variability and base models. The resolved model is essentially the product architecture without variability that the system deploys in each device.

Reconfiguration. Once deployed in the system's devices, the initial product configuration might require modification as part of the system's self-adaptation to context changes. Self-adaptation has a number of process steps:

- *Specify new context values as new constraints*. In the location-based service example, the adapted product constraint is nSat = 6.
- *Add new constraints*. These are the elements selected in the running product's resolution model. In the example, the new running product constraints are GSM and Wi-Fi.
- *Create new resolution models*. These models incorporate the new context value and running product constraints as well as the variability model constraints. If any new context values are incompatible with resolution model elements, remove them (the example has no incompatibilities).
- *Generate all resolution models compatible with this new context*. Any possible combination of the four location technologies in Figure 3 would work well with the new context.
- *Select the best resolution*. Among the resolutions that satisfy the context, evaluate QoS using constraint-satisfaction problem (CSP) techniques.[8]
- *Execute the best resolution model in CVL*. The result is the adapted product's resolved model.
- *Transform the running product into the adapted one at implementation level*. For this purpose, FamiWare implements an operator that obtains the differences between two variability model configurations with cardinality.

- *Generate a reconfiguration plan*. The plan, written in a reduced version of OWL-S, contains the implementation-level tasks to perform the changes and the correspondence between the variability model modifications and the implementation tasks.
- *Interpret the generated plans*. FamiWare's reconfiguration service executes the corresponding tasks. The code associated with every task must already be implemented, but to add tasks, developers simply provide the code, and FamiWare automatically augments its product line.

If the reconfiguration time isn't critical, a slower reconfiguration will suffice.

Because our approach uses models@run.time, the product running in each device corresponds to the resolved model that satisfies every context. The generated plans respect the constraints between variable elements, which ensures that the global system enters a valid state after reconfiguration. FamiWare defines the dependencies between the different mobile devices as constraints between clones in the variability model, which ensures that the running product after reconfiguration is still compatible with the devices.

The strong advantage of our approach is that only reconfiguration plans, at most 4 Kbytes, are loaded in the devices and sent over the network, not the variability model, which is roughly 10 times larger.

Application to Dynamic Route Planning

To evaluate our self-adaptation process, we applied it to a dynamic route-planning service that runs in the highway control center of an intelligent transportation system. The service, which is part of the EU's Interoperable Trust Assurance Infrastructure (Inter-Trust) Project,[9] calculates the fastest route given a vehicle's current position and the desired destination.

The service considers user preferences (QoS) and contexts such as weather or rush-hour traffic estimates, selects an optimal route, and sends it to the vehicle's onboard unit. Periodically, the unit's location-based service resends the vehicle's position to the control center to inquire if the route needs recalculating. Events such as a connectivity loss might trigger route recalculation.

The onboard unit's location-based service might have to self-adapt to correct its inability to determine or communicate the vehicle's position. It might also need to optimize functions. QoS considerations include reconfiguration time, communication encryption limits, energy use versus accuracy, and reconfiguration cost.

Reconfiguration Time

The system must detect at least 95 percent of all system faults and take no longer than 10 minutes to detect a full service failure. At some point, the control center's context-awareness service realizes that it isn't receiving a vehicle's position, which implies that the onboard unit's location-based service is malfunctioning. The service might be using a GSM/3G network when the vehicle enters a heavy traffic zone. Because this network type is highly susceptible to bandwidth limitations from overcrowding, the location-based service could become unavailable.

To continue providing position information, the system must reconfigure. It could use GPS or Wi-Fi to determine position and short-range communication, such as roadside units, Wi-Fi, or geonetworking to send the position to the control center. The best alternative will depend on the required QoS.

For example, in an emergency, vehicle position can be vital, so the configuration with the lowest reconfiguration time and highest accuracy will be best. GPS takes time to find enough satellites, and geonetworking is slow, so the system discards those choices and selects Wi-Fi as the best option for determining and sending the vehicle's position.

The reconfiguration plan, P, for this option consists of four tasks: turn off the GSM/3G module, activate Wi-Fi, use Wi-Fi to get the position, and use Wi-Fi for the communication module (to send the position). In CVL, this plan is

$$P = \{\text{Deactivate (GSM)}, \text{Activate (Wi-Fi)}, \text{GetPosition (Wi-Fi)}, \text{Communication (Wi-Fi)}\}.$$

On the other hand, if the reconfiguration time isn't critical, a slower reconfiguration will suffice, and either geonetworking or GPS will be the best option. The reconfiguration plan then involves not only the user's onboard unit but also the units in ve-

hicles nearby to enable geodissemination (message dissemination confined to a particular geographic region).

To compare reconfiguration time for the emergency and noncritical scenarios, we measured time from context change, the point at which the control center realizes it isn't receiving the vehicle's position, to the end of reconfiguration.

It took 58 seconds for the context-awareness service to choose the new resolution models and select the one that best fit the required QoS. The average reconfiguration in an emergency situation was 75 s; in the noncritical scenario with geodissemination, the average was 189 s, primarily because reconfiguration involved more devices. Even so, the total reconfiguration time of 247 s for that scenario was still well under the fault-detection requirement of 10 min. The total time in the emergency scenario was only 133 s.

As we expected, reconfiguring the system incurs an overhead penalty, but the compensation is that the onboard unit won't fail to determine or communicate vehicle position, which in an emergency could be a vital function.

Secure Communication Latency

Communication between the control center and onboard unit and between the roadside unit and onboard unit must be secure. However, encryption, decryption, and any supplementary security functionality must not exceed 500 ms for each roundtrip message. The QoS related to latency might make some encryption strategies more suitable than others.

Energy-Accuracy Tradeoff

Managing battery life is crucial for mobile devices, since services such as GPS have a high energy expenditure. When the battery drops below a certain level, the location-based service can reconfigure to save energy. One option is to read and send the GPS position less frequently or use other technologies for those functions.

However, selecting a configuration requires more than considering energy savings. In the emergency scenario, the reported position must be accurate within 100 meters. Consequently, when more than four satellites are visible, GPS is required, regardless of energy expenditure. For the noncritical scenario, position accuracy can be up to 200 meters, so GSM would be good enough.

Our approach uses CSP to evaluate tradeoffs between energy expense and accuracy.

Reconfiguration Cost

Each system adaptation implies a reconfiguration cost in both time and wasted energy, for example, when messages with large code chunks are sent over the network. To consider reconfiguration cost, providers can simulate before runtime possible energy savings with new products, thereby choosing only those that save more energy than the reconfiguration process expends.

Mobile system self-adaptation is far from trivial, requiring enough flexibility to accommodate highly diverse devices, QoS requirements, and contexts. Our approach uses variability models to drive reconfiguration

ABOUT THE AUTHORS

NADIA GÁMEZ is a postdoctoral researcher in the Component and Aspect-Oriented Software Development (CAOSD) Group in the Software Engineering Group at the University of Malaga (GISUM). Her research interests include software product lines, ambient intelligence, mobile and embedded systems, and wireless sensor networks. Gámez received a PhD in computer science from the University of Malaga. Contact her at nadia@lcc.uma.es.

LIDIA FUENTES is head of the CAOSD research group (http://caosd.lcc.uma.es) and professor of computer science at the University of Malaga. Her research interests include software product lines, component-based middleware, aspect-oriented software development, software agents, and ambient intelligence. Fuentes received a PhD in computer science from the University of Malaga. Contact her at lff@lcc.uma.es.

JOSÉ M. TROYA is head of GISUM (www.gisum.uma.es) and a professor of computer science at the University of Malaga. His research interests include optimization problems; parallel algorithms; software engineering in distributed and real-time systems; and software architecture, methodology, and programming languages for critical systems. Troya received a PhD in physics from the Complutense University of Madrid. Contact him at troya@lcc.uma.es.

but is both nonintrusive, since variability is expressed outside the base model, and lightweight, since large models aren't loaded into the devices.

The use of DSPL for pervasive applications isn't new, and researchers have proposed elements of our approach. Some have shown that autonomy is possible with variability models[10] but use feature models to guide reconfiguration, not a simple language like CVL. Others propose a mechanism to change structural variability of embedded systems at runtime,[11] but it lacks a generic self-adaptation framework like FamiWare. CANDEL is a generic framework for representing context information as a dynamic product line of context primitives,[12] but it has no model-driven reconfiguration process. Finally, through architectural models executed at runtime, Madam enables generic middleware components to carry out mobile system self-adaptation of mobile systems.[13] However, there's no language to describe variability outside the base models.

In short, none of this work considers the breadth of requirements we've identified. Moreover, we've implemented our work for sensors as well as smartphones and thus recognize the importance of not overloading tiny devices with large models. Our CSP-based approach to maintaining QoS is highly flexible, allowing runtime tradeoffs in time and optimizations.

Acknowledgments

This work was supported by the European INTER-TRUST FP7-317731 project and the Spanish TIN2012-34840, FamiWare P09-TIC-5231, and MAGIC P12-TIC1814 projects.

References

1. S. Hallsteinsen et al., "Dynamic Software Product Lines," *Computer*, vol. 41, no. 4, 2008, pp. 93–95.
2. M. Hinchey, P. Sooyong, and K. Schmid, "Building Dynamic Software Product Lines," *Computer*, vol. 45, no. 10, 2012, pp. 22–26.
3. N. Bencomo et al., "A View of the Dynamic Software Product Lines Landscape," *Computer*, vol. 45, no. 10, 2012, pp. 36–41.
4. G. Blair, N. Bencomo, and R. France, "Models@run.time," *Computer*, vol. 42, no. 10, 2009, pp. 22–27.
5. N. Gámez and L. Fuentes, "FamiWare: A Family of Event-Based Middleware for Ambient Intelligence," *Personal and Ubiquitous Computing*, vol. 15, no. 4, 2011, pp. 329–339.
6. O. Haugen et al., "Adding Standardized Variability to Domain Specific Languages, *Proc. 12th Int'l IEEE Software Product Line Conf.* (SPLC 08), 2008, pp. 139–148.
7. N. Gámez and L. Fuentes, "Architectural Evolution of FamiWare Using Cardinality-Based Feature Models," *J. Information and Software Technology*, vol. 55, no. 3, 2013, pp. 563–580.
8. N. Gámez et al., "Constraint-Based Self-Adaptation of Wireless Sensor Networks," *Proc. 2nd ACM Int'l Workshop Adaptive Services for the Future Internet and 6th Int'l Workshop Web APIs and Service Mashups* (WAS4FI-Mashups 12), 2012, pp. 20–27.
9. A. Badii and D. Thiemert, "Requirements Specification, Inter-Trust Project Deliverable D2.1.1," 2013.
10. C. Cetina et al., "Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes," *Computer*, vol. 42, no. 10, 2009, pp. 37–43.
11. J. Bosh and R. Capilla, "Dynamic Variability in Software-Intensive Embedded System Families," *Computer*, vol. 45, no. 10, 2012, pp. 28–35.
12. Z. Jaroucheh et al., "CANDEL: Product Line Based Dynamic Context Management for Pervasive Applications," *Proc. IEEE Complex, Intelligent and Software Intensive Systems* (CICIS 10), 2010, pp. 209–216.
13. J. Floch et al., "Using Architecture Models for Runtime Adaptability," *IEEE Software*, vol. 23, no. 2, 2012, pp. 62–70.

Continued from p. 116

takes a month or two to get just a single machine. The real thing that makes it cloud computing is self-service. You make an API call, and a few minutes later a bunch of machines turn up. That is the most fundamental difference. It speeds up the whole procurement cycle. It makes everything much more dynamic.

You can use the cloud as a faster way to do the things you used to do in datacenters. But the really interesting things come when you start realizing what happens when you use it in a much more dynamic way by using machines as ephemeral resources. You can turn them on, turn them off, use a machine for a couple of hours, and then give it back. That is the essence of cloud. It comes down to putting self-service tools in the hands of the developers to do things themselves, rather than making "what operations used to do" into an API.

Developers don't care whether it's a public cloud like AWS [Amazon Web Services] or a private cloud inside the company, as long there's enough capacity for whatever you need to do. It's just there.

How did Netflix end up moving to the cloud?

Netflix started off as a DVD shipping company. It wasn't even seen as a very big technology company at the time. When I joined, its personalization algorithms were considered its primary interesting technology, not its scale.

Netflix had around 6,000,000 customers when I joined in 2007. Typically, every weekend the customers would visit the website, decide what DVDs they wanted to have shipped in the next week, and prioritize by shuffling their queue. Every time they sent a disc back, we would send them another one. The interaction with Netflix was sending a disc in through the postal service. That required a few tens of Web servers, a few back-end machines, and a big database running on a monolithic centralized app.

That year, we launched streaming with a very small catalog, but it started to take off quite quickly. When you interact with a streaming service, every click goes to the website. You browse around the website. When you decide that you want to watch something, you click Play. Then traffic goes back and forth figuring out what to do: finding the machine, finding the movie, doing authorization, giving you a security key to decode that movie, and then logging the activity so that we can make sure that there's good quality of service (rebuffers, calculating the speed to run at, and determining which content delivery network to use).

There are enormously many transactions to the back end. Streaming generates around a thousand times more Web requests than the DVD service. That was fine when we were just starting out with a small number of machines, a small number of movies, and not many customers using it. But the usage rate started to increase very rapidly because there was nothing stopping customers from watching a lot of movies. They no longer had to send a DVD back and wait for another one. So, we started getting a lot of people binge-watching.

The number of interactions people had with Netflix, the number of things they watched, and the number of interactions with the website per view all went up by orders of magnitude. Our datacenter consisted of a couple of small machines in the corner that were put in to initially launch streaming. And they were running out of capacity incredibly quickly. In 2008, there was a big outage when the central monolithic app broke due to a storage problem (a corruption in the storage area network corrupted Oracle). It was a big mess.

As a result, we decided we were not very good at running stuff in the datacenter. We started thinking about how to scale for this incredible future workload, where we didn't know how fast it would grow. And that really comes down to the core of why this is interesting: because we could not predict how much capacity we would need.

In 2009, we moved some of the back-end batch workloads like encoding movies to the cloud. In 2010, we moved the front-end website to the cloud. In 2011, we moved the database back end so that the master copies of all the data were in the cloud, and in 2012, we started open-sourcing the tooling we'd built to do that. Those are the main history points.

You've hinted that software architecture changes when you move to the cloud. What changes?

You can use cloud as nothing more than a faster way to do datacenter stuff, but that misses most of the benefit. The real benefit comes when you start doing things that you couldn't have done in your datacenter. You can trivially do hardware experiments that last a few days on a huge scale, scattered all over the world, something that you wouldn't even think of doing if you had to tell your ops guys, "Hey, I need a liberally distributed database with a hundred nodes in it and a couple hundred terabytes of solid state disc. And I'd like it this afternoon, in six different datacenters. And whatever."

We did this and we did it without asking permission. It took about

SOFTWARE ENGINEERING RADIO

Visit www.se-radio.net to listen to these and other insightful hour-long podcasts.

RECENT EPISODES

- 217—Charles Anderson talks with James Turnbull, the creator of Docker, a popular lightweight Linux deployment tool. Somewhere between a process and a virtual machine, Docker is emerging as a container for isolating microservices.
- 218—Robert Blumen discusses the Command-Query-Responsibility Segregation (CQRS) architectural pattern with Udi Dahan, one of the pattern's cocreators. CQRS formally separates a distributed system into a write master and one or more read models.
- 219—Jeff Meyerson interviews Apache Kafka committer Jun Rao on the high-throughput distributed event bus that combines features of messaging and publish–subscribe.

UPCOMING EPISODES

- 220—Robert Blumen sits down in person with Jon Gifford for a conversation about logging, APIs, log record formats, and how search engines are transforming the collection and interpretation of program messages.
- 221—Johannes Thönes and chief guru Jez Humble converse on the origins of continuous delivery (CD) in the lean movement, how to build a CD culture, and how to introduce CD in regulated environments.
- 222—Apache Storm Founder Nathan Marz chats with Jeff Meyerson about stream processing, "real-time Hadoop," the lambda architecture, and thinking about streaming in terms of spouts and bolts.

20 minutes to create. We built a very write-intensive globally distributed database to see what would happen. The decision to do it was made while we were walking out of a meeting. The guy who did it wandered by somebody else's cube and asked that person to set it up. That afternoon it was created. We put 18 Tbytes of data from backup on it. And then we hammered the thing as hard as we could with all kinds of error and failure injections to make sure it worked well. A few days later we removed it.

You couldn't do that if you had real infrastructure because it would take too long to get approval. In a datacenter, you would need a multimillion-dollar machine. I didn't know in advance what it would cost, but it worked out to a few hundred dollars per hour. The value we got out of it was much greater: at a meeting the following week, we said, "We just proved this works."

At the time, there was an internal argument going on about how we would build distributed systems and whether we could rely on high bandwidth in a global cloud. When you go to a meeting with working code or benchmarks, you win arguments. That short-circuited an enormous amount of what would have been debate and justification.

Can you explain the Chaos Monkey?

I will explain the principle, and then it will be obvious that it makes sense. There is an analogy of cattle versus pets. If you know your machines in production by name, and if one goes down everyone gets upset, then that's a pet. You have to take it to the vet if it gets sick. The other kinds of machines are cattle. When you have a herd of cattle in the field, they produce so many gallons of milk. If a few of them die, you get slightly less milk that day, and you buy some more cattle.

The principle that Netflix adopted was that everything in production is a herd of cattle. There are no pets, no individual machines that if one went down anybody would care about. Everything is on an autoscaler, even a single machine.

Once you have established that principle, then you have to test that compliance by killing individual machines chosen at random. That is what the Chaos Monkey does. It picks a random time to stop some machines. The autoscaler should automatically replace it. If somebody snuck an individual machine into production, the Chaos Monkey killed it, and they got upset, well, they should not have done that, right? It forces the developers to think in the new way. Everything you launch is on an autoscaler even if there is only one machine in the group. It must be a stateless, disposable machine that can be restarted.

They took it to the next level

with the data layer, which is a triple-replicated Cassandra back end. The Chaos Monkey kills those as well—including the discs that are inside the instances. You might lose a few hundred gigabytes of data when you delete the instance. It's not attached storage: the disks are inside the instance. But it's replaced as the data is resynchronized from the other two copies, proving that you can build an ephemeral data layer as well.

This is a different way of thinking from a sort of datacenter mentality where machines should always stay up. You would use perfect machines if you could. Instead, you create herds of machines that are extremely resilient because you can lose large numbers of them and everything still works.

Let's talk about the CAP (consistency, availability, partition tolerance) theorem. Which part of it do you apply? Which side of the triangle do you lean to?

You have to decide whether consistency or availability is most important to you. The basic principle of Netflix is that no partition or failure should take out the service. We lean very heavily to the availability side when things are partitioned, which means that if you slice Netflix up and drop all the networks between all the different parts of the system, the system continues to work. The isolated parts will just carry on working. And they'll gradually become inconsistent with the rest of the system. When you reconnect, "last writer wins" takes over. Cassandra kicks in. Whoever wrote a piece of data last ends up overwriting whatever was written in the meantime.

The system gradually gets back to being consistent, although you may lose a few updates if you modified something that was modified somewhere else. Generally it doesn't, and anyway, it's better to deal with inconsistency than to be down. For a service like Netflix, where 50 million people are trying to watch movies around the world, they have an expectation that when they turn on a TV set, it should just work. There should never be a message saying, "We're down right now."

It's hard to tell a three-year old that they can't watch their dinosaur cartoons because Netflix has failed.

Yeah, tell me about it.

STEFAN TILKOV is cofounder and principal consultant at innoQ, a technology consulting company with offices in Germany and Switzerland. Contact him at stefan.tilkov@innoq.com.

IEEE Software (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., Los Alamitos, CA 90720; +1 714 821 8380; fax +1 714 821 4010. IEEE Computer Society headquarters: 2001 L St., Ste. 700, Washington, DC 20036. Subscribe to *IEEE Software* by visiting www.computer.org/software.

Postmaster: Send undelivered copies and address changes to *IEEE Software*, Membership Processing Dept., IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854-4141. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8, Canada. Printed in the USA.

SOFTWARE ENGINEERING

Editor: **Robert Blumen**
Symphony Commerce
robert@robertblumen.com

AUDIO

The Modern Cloud-Based Platform

Stefan Tilkov

THIS MONTH'S EXCERPT from Software Engineering Radio (www.se-radio.net) features the third in a recent series of podcasts touching microservices, beginning with *210: Stefan Tilkov on Microservices* and *213: James Lewis on Microservices*. Docker was the subject of the fourth podcast, episode 217, in which Charles Anderson tracked down that project's founder for an in-depth discussion.

Netflix established itself as a company by disrupting the video rental industry with monthly pricing, a deep back catalog, and machine-learning-based recommendations. But it's barely in the DVD business now. The second Netflix revolution has been an aggressive move into streaming content from the Amazon cloud. Netflix and its US customers are one of the heaviest users of the US Internet, by some estimates accounting for over one-third of all traffic during peak movie-watching hours. Its movie-streaming business has been the source of technological innovations as it has innovated to meet consumers' ever-rising standards.

Much of what Netflix has learned is now public information through the Netflix open-source stack and the tireless efforts of Adrian Cockroft (formerly of Netflix), one of the major cloud architects during his years there. Cockroft has been a regular speaker at tech conferences, many of which are online.

In SE Radio episode 216, Cockroft and our newest host, Stefan Tilkov, converse about Netflix's move to the cloud, development speed as the critical competitive factor, how the monolith gave way to microservices, microservices at scale, microservices and DevOps, the Netflix service discovery infrastructure, distributed debugging in a deep microservices stack, geographic redundancy on the Amazon cloud, being always on, availability over consistency, the strategic plan behind open source, how open source helps with hiring in a competitive market, and what Adrian is doing post-Netflix in the venture-capital field.

We would enjoy hearing from readers of this column and listeners to the podcast. We accept incoming emails at se-radio@computer.org and tweets and direct messages to @seradio. You can also visit our Facebook page, Google+ group, and LinkedIn group. To hear this interview in its entirety, visit www.se-radio.net. —Robert Blumen

It's hard to read any sort of article these days that doesn't somehow mention the cloud or cloud computing. Despite that, how do you define those things? What is the cloud, and what does cloud computing mean to you?

The biggest change is when someone working at a company thinks, "I need some machines to do something." You have to file a ticket and wait for somebody else to get around to sorting out that ticket. In some big companies, it

Continued on p. 113

0740-7459/15/$31.00 © 2015 IEEE

39th Annual International Computers, Software & Applications Conference

www.compsac.org

CALL FOR PAPERS

July 1-5, 2015
Tunghai University
Taichung, Taiwan

Mobile and Cloud Systems - Challenges and Applications

COMPSAC is the IEEE Signature Conference on Computers, Software, and Applications. It is one of the major international forums for academia, industry, and government to discuss research results, advancements and future trends in computer and software technologies and applications. The technical program includes keynote addresses, research papers, industrial case studies, panel discussions, fast abstracts, doctoral symposium, poster sessions, and a number of workshops on emerging important topics. With the rapidly growing trend in making computations and data both mobile and cloud-based, such systems are being designed and deployed worldwide. However, there still exists several challenges when they are applied to different domains or across domains. COMPSAC 2015 will provide a platform for in-depth discussion of such challenges in emerging application domains such as smart and connected health, wearable computing, internet-of-things, cyber-physical systems, and smart planet.

Technical Symposia | **Workshops Program** | **Special Sessions**

COMPSAC 2015 will be organized as a tightly integrated union of several symposia, each of which will be focusing on a particular technical segment. **Please visit www.compsac.org for full information on symposia organization.**

* **Symposium on Embedded & Cyber-Physical Environments**
* **Symposium on Software Engineering Technologies & Applications**
* **Symposium on Technologies and Applications of the Internet**
* **Symposium on Security, Privacy and Trust Computing**
* **Symposium on Mobile, Wearable and Ubiquitous Computing**
* **Symposium on Web Technologies & Data Analytics**
* **Symposium on Human-Machine and Aware Computing**
* **Symposium on Novel Applications and Technology Advances in Computing**
* **Symposium on Computer Education and Learning Technologies**
* **Symposium on IT in Practice**

Authors are invited to submit original, unpublished research work and novel computer applications in full-paper format. Simultaneous submission to other publication venues is not permitted. The review and selection process for submissions is designed to identify papers that break new ground and provide substantial support for their results and conclusions as significant contributions to the field. Submissions will be selected that represent a major advancement in the subject of the symposia to which they are submitted. Authors of submissions with a limited contribution or scope may be asked to revise their submissions into a more succinct camera-ready format; e.g., a short paper, workshop paper, fast abstract, or poster.

COMPSAC 2015 will also feature a workshops program for topics closely related to the conference theme, *Mobile and Cloud Systems - Challenges and Applications.* Special sessions such as Fast Abstract and Industry Papers will be applicable especially for researchers and engineers who would like to present a new, early and work-in-progress ideas, method, and analysis. The Doctoral Symposium will provide a forum for doctoral students to interact with other students, faculty mentors, industry and government. Students will have the opportunity to present and discuss their research goals, methodology, and preliminary results within a constructive and international atmosphere.

Important Dates for Authors:

January 17, 2015: Paper submissions due
March 15, 2015: Paper notifications
April 28, 2015: Camera ready and registration due

Contact COMPSAC

For full information and CFP please visit www.compsac.org
Contact COMPSAC organizers at cs_compsac@iastate.edu

COMPSAC Sponsors:

IEEE computer society | IEEE | IEEE CLOUD COMPUTING

COMPSAC Technical Co-Sponsors:

Information Processing Society of Japan | CCF | Korean Institute of Information Scientists and Engineers

COMPSAC 2015 Local Host:

東海大學 TUNGHAI UNIVERSITY