2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

Modern Release Engineering in a Nutshell

Why Researchers should Care

Bram Adams
Polytechnique Montréal, Canada
bram.adams@polymtl.ca

Shane McIntosh
McGill University, Canada
shane.mcintosh@mcgill.ca

***Abstract*—The release engineering process is the process that brings high quality code changes from a developer's workspace to the end user, encompassing code change integration, continuous integration, build system specifications, infrastructure-as-code, deployment and release. Recent practices of continuous delivery, which bring new content to the end user in days or hours rather than months or years, have generated a surge of industry-driven interest in the release engineering pipeline. This paper argues that the involvement of researchers is essential, by providing a brief introduction to the six major phases of the release engineering pipeline, a roadmap of future research, and a checklist of three major ways that the release engineering process of a system under study can invalidate the findings of software engineering studies. The main take-home message is that, while release engineering technology has flourished tremendously due to industry, empirical validation of best practices and the impact of the release engineering process on (amongst others) software quality is largely missing and provides major research opportunities.**

I. Introduction

Release engineering is the process responsible for taking the individual code contributions of developers and bringing those to the end user in the form of a high quality software release. From start to finish, a myriad of tasks need to be performed by an organization's *release engineers*, i.e., the personnel whose main duties are the development, maintenance, and operation of an organization's release infrastructure.

Broadly speaking, the patches of developers need to be reviewed and integrated (merged) [13] from developer branches into team and product branches, where they are compiled [48] and tested [88] by the Continuous Integration (CI) system, until they land in the master branch. When an upcoming release enters the feature-complete stage, release stabilization begins [60], where major bugs in the functionality of the release are addressed. When the release date is near, the new release needs to be deployed [22], i.e., copied to a web server, virtual machine or app store. Finally, the deployed release needs to be made available ("released") to users [86].

In contrast to other research fields (even within the software engineering discipline), recent advances in release engineering have largely been made by industry, not academia. Indeed, it was market pressure, the influence of agile development and the desire to bring value to the customer faster that inspired the recent phenomenon of continuous delivery [36]. For example, while the releases of the past would take months or even years to produce, modern applications like Google Chrome [69], Mozilla Firefox [70] and the Facebook Mobile app have a release "cycle time" of 2-6 weeks, while web-delivered content like the Netflix and Facebook websites push new releases 1-2 times daily [65]. Furthermore, lean web apps like the popular IMVU chat application[1] release up to 50 times per day [26].

As these pioneering organizations successfully developed experimental tools and practices to make such rapid release cycles a reality, Facebook's release engineering director, Chuck Rossi, claimed that "continuous delivery for web apps is a solved problem" [65]. However, he did add "..., yet continuous delivery for mobile apps is a serious challenge." Indeed, for every success and breakthrough that has been made, there are a slew of failures. Even today, software organizations who are not at the forefront of the release engineering revolution need to consider what release practices should be adopted, what tools they should invest in and what profiles should be used to make hiring decisions. Even for the pioneers of modern release engineering, newer technologies like mobile apps still pose open challenges. Broader questions include: what will be the long-term effect on user-perceived quality of releases [43, 45], how quickly will technical debt ramp up when release cycles are so short and can end users keep up with a continuous stream of new releases?

While most of the recent advances in release engineering, such as best practices and tooling, have been driven by industry, researchers can play an essential role in addressing the above questions. For example, researchers can provide value by analyzing release engineering data stored in industrial version control systems, bug/review repositories, CI servers and deployment logs, with the aim of gleaning actionable insights for release engineers. Such insights could help them decide what best practices to adopt, understand why (and when) continuous delivery is feasible, and what compromises are necessary in terms of software quality and technical debt.

As a first step towards promoting release engineering research, this paper makes three major contributions:

1) Providing researchers with a working definition of modern release engineering pipelines and their repositories.
2) Discussing each release engineering activity in more detail, proposing promising avenues for research that, in our opinion, can help release engineers today.

[1] http://www.imvu.com/

978-1-5090-1855-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SANER.2016.108

3) Providing a checklist of three release engineering decisions that can impact software engineering studies. Researchers should consider this checklist to avoid wrong conclusions, regardless of whether their study targets the release engineering process!

II. A Working Definition of the Release Engineering Pipeline

Fig. 1 provides an overview of the typical phases in a modern release engineering pipeline, adapted from the concepts discussed in seminal release engineering books [9, 13, 36]. This section will discuss the role and state-of-the-practice for the 6 major phases indicated in the overview. We will also mention the most common tools of each phase, but for a more complete listing of tools at the time of writing, we refer elsewhere [90]. Finally, we consider the later testing stages outside the scope of this work, since testing is a separate domain of its own, however we will discuss the main release engineering challenges related to testing within the CI phase.

While we do discuss recent research advances with respect to the 6 major phases, our goal is not to provide a complete survey of existing research on release engineering. For such an in-depth analysis, we refer to Dearle [22], Mäntylä et al. [45], Rodríguez et al. [64] and Rahman et al. [59].

A. Integration: Branching and Merging

The first phase in the release engineering pipeline is the movement of code changes made by developers in their own development branch across their team's branch all the way up to the project's master branch [13]. Past work has explored how successful organizations allow code changes to enter this flow [15, 38] and how code changes flow once they have entered it [5, 27].

Bringing high quality code changes as fast as possible to the project's master branch, without reducing code quality [71], is a key concern for software organizations. To achieve this goal, software teams rely on Version Control Systems (VCSs) like `Subversion` or `Git`, which store subsequent revisions of each manually created and maintained file. A branch can be created starting from a revision of some parent branch (e.g., master branch), after which the branch's team of developers records a chronological sequence of code changes ("commits") on the branch, for example to fix a bug or add a new feature. Those commits are invisible to other branches (teams) until the team decides to "merge" the branch back into its parent branch. Such a merge allows a branch's commits to flow to the parent branch, effectively ending the commits' isolation.

Since commits typically flow from cutting-edge development branches to more strictly controlled release quality branches, quality assurance activities like code reviews are essential before doing a merge or even allowing a code change to be committed into a branch [8, 11, 62, 63]. Indeed, recent work explores what kinds of issues are found and discussed during these reviews [12, 46], and also studies the impact that these reviewing practices have on software quality [39, 44, 49, 51, 52, 81].

Despite careful reviewing of individual commits, branch merges still pose a risk of so-called merge conflicts [18, 66], which cause significant costs for companies [17]. Basically, while a team is working in isolation in its branch, other teams are doing the same in their own branches. The longer that this parallel development takes before merging, the more code changes have been merged in the meantime into the parent branch, and the more incompatible both branches will have become. For example, other teams could have changed the same physical code line in the same file (text-level conflict), could have removed or renamed essential methods (build-level conflict), or could have made other changes that either cause test failures (test-level conflict) or failures during regular execution (semantic conflict). These merge conflicts can introduce delays in product development, requiring developer effort to thoroughly understand the code changes in multiple branches in order to combine them correctly.

When the developers of a branch eventually attempt to merge back their commits, of which they have no idea whether they conflict with other teams' commits, the VCS will warn them of text-level conflicts. In fact, the VCS will refuse to continue merging as long as these textual conflicts are not resolved. However, unless the developers compile, test and execute their branch after resolving the textual conflicts and before finishing the merge, they will not be warned about compilation, test and semantic conflicts. These conflicts can prevent the resulting system from compiling or executing correctly, effectively breaking the system for other development teams and causing delays [34, 41, 68].

There are rules of thumb that mitigate the risk of introducing merge conflicts. The best one is to keep branches short-lived and merge often, since this reduces the time period during which other teams could have made conflicting code changes. If this is impossible because the features or fixes being developed in a branch are not yet ready, one can also "rebase" ("sync up") a branch [16] with the recent changes of the parent branch, as if one would merge the parent branch into the child branch. Any conflicts caused by the sync-up can be resolved immediately inside the child branch, substantially reducing the potential for merge conflicts by the time the final merge of the child branch into the parent branch will happen. To help assess build and test conflicts, many projects also provide "try" servers to development teams (e.g., [82]), which provide a mechanism for running the CI build and test processes (see next section) on-demand before performing a merge.

To reduce the burden of branches, merges and conflicts, companies have recently adopted "trunk-based development," which eliminates most (if not all) branches below the master branch. Instead, teams directly commit to the master branch (after review). Since this eliminates the safe isolation offered by branches, a different coordination mechanism is needed to avoid impacting active development of other teams with incomplete features. Such incomplete changes are said to "break trunk," i.e., causing the most recent commit on master branch (from which every developer will start his or her

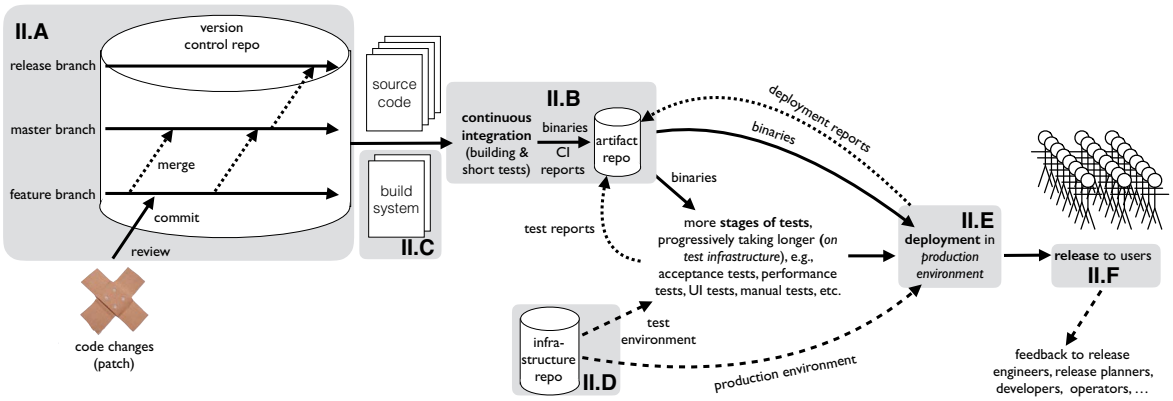

Figure 1: An overview of a modern release engineering pipeline [9, 13, 36], and how it maps to the subsections of the paper.

work) to fail compilation or testing. To avoid trunk breakage, teams require a means to indicate that certain features are "in progress," while others are "under test" or "permanent."

Feature toggles [9] are the mechanism that modern companies (e.g., Facebook and Netflix) rely on to provide isolation in the absence of branches (we will see other uses of feature toggles in the deployment phase). Toggles allow the source code of incomplete features to be put inside a conditional block that is controlled by means of a variable (i.e., a "toggle") whose name refers to the feature that is under development. Teams can use conditional compilation for this or regular if-conditions using global variables. If the toggle's value is set to "true," the feature should be compiled and tested. Otherwise, the feature is disabled and, as long as the code inside the conditional block compiles cleanly (in case of if-conditions), nobody is affected by the incomplete feature.

B. Continuous Integration: Building and Testing

Continuous Integration (CI) refers to the activity of continuously polling the VCS for new commits or merges, checking these revisions out on dedicated build machines, compiling them and running an initial set of tests to check for regressions [25, 73]. CI tools like `Jenkins`, `Bamboo`, `Team Foundation Server` or `Buildbot` enable teams to specify the VCS to monitor, as well as which build commands to run. A dashboard provides a visual overview of the status of each commit (i.e., which commits have introduced regressions), while interested practitioners can subscribe to notifications of broken builds and failed tests. Recent empirical studies suggest that the rapid feedback loop provided by CI has a positive effect on team productivity, while not significantly impacting code quality [87].

Since CI begins immediately after a commit or merge has been made, feedback needs to be fast to allow the developer (or team) responsible for the commit or merge to resolve the breakage while his or her changes are still fresh in their mind. To keep CI builds short, CI typically does not run a full test suite after compilation, but a representative subset. The idea is that the rest of the test suite, as well as tests that take more time, such as integration, system or performance tests, will be run in later stages, typically at set intervals such as nightly or on weekends. Failures of such tests typically do not indicate an issue of a small commit, but rather of a larger set of functionalities, which is why a lower frequency of the execution of such tests makes sense. Although accelerating these slower tests will be appreciated by development teams, faster CI tests will likely have a bigger impact.

Apart from slow CI tests, another major reason for slow CI builds in organizations with large development teams, is the simultaneous submission of multiple commits, yielding a large number of new commits and merges in a short timespan. Just having one CI server perform a separate CI build for each commit does not work in practice, as the server will become overburdened. One should either deploy multiple CI servers to share the load or execute builds for groups of commits together, e.g., instead of running 3 separate builds for commits A, B and C, one would do just one build to test the 3 commits together. Although this theoretically speeds up the CI process by a factor of 3, this speed-up only holds in the case that the build is successful. If any of the commits has a bug, the build can fail, but without a clear idea which commit is responsible for the breakage. Hence, one typically needs to redo the builds for the 3 commits separately to identify the culprit, which causes the total build time to be higher than doing all 3 commits in parallel from the start.

Since the later test stages, such as performance testing, should use exactly the same deliverables (binaries) as those produced by the CI process, it should not be necessary to recompile those before each test stage. Instead, all test stages obtain the deliverables that were built by the CI process from a so-called "artifact repository." Essentially, this is a file server that can tag files such as binaries, libraries or any kind of document (e.g., test reports) with metadata like commit IDs or release names. As soon as a bug is reported in a particular release, or a test failed in a particular CI build, an artifact repository tool like `Artifactory` or `Nexus` is able to swiftly locate the correct deliverables that went into the release or build in question.

C. Build System

The build system is the set of build specification files used by the CI infrastructure (and developers) to generate project deliverables like binaries, libraries or packages (e.g., for an operating system distribution like Debian) from the source code. Moreover, the build system automates many other activities, such as test execution and sometimes deployment. A build system typically consists of a configuration layer and a construction layer [3]. The configuration layer is used to select which features should be compiled and included in the resulting deliverables, as well as which build tools (e.g., compilers, interpreters) are necessary to compile those features. Once configured, the construction layer is used to specify the build tool invocations that are required to generate deliverables from source code. Since these build tool invocations are order-dependent (e.g., C++ files must first be compiled before the resulting object files can be linked together), a key responsibility of the construction layer is to invoke build tools while respecting their dependencies.

The build systems of large, multi-platform software systems are complex and difficult to understand and maintain. To assist in this regard, research has proposed tools to visualize and reason about build systems [4, 79, 83]. Furthermore, recent work has begun to explore how build system problems are being addressed [7, 21, 33, 53, 54].

Hundreds of different build tools exist, written for different programming languages and using different paradigms. Older programming languages like C and C++ have file-based build dependencies, where each file is compiled separately and dependencies are declared between individual files. `GNU Make` is the most popular file-based build system technology [50]. Java-based systems have task-based build dependencies, where a compiler or other tool is able to compile multiple source code files at once. `Ant` is the prototypical task-based build system technology. For example, a `compile` task of an Ant build system would invoke the Java compiler, which may read several `.java` files and update several `.class` files.

Lifecycle-based build technologies like `Maven` consider the build system of a project to have a sequence of standard build activities that together form a "build lifecycle." By placing source code files in the right directory, tools like Maven can automatically execute the right lifecycle build activity. Finally, build system generators like `CMake` provide a high-level domain-specific language to specify a build, which is then transformed automatically into build files of one of the three other build system technologies. Often, specification files can be generated for different operating system platforms to ease the process of developing a cross-platform build system.

D. Infrastructure-as-Code

One of the most recent innovations in the release engineering pipeline is infrastructure-as-code [36]. The term "infrastructure" (or "environment") refers to the server, cloud, container or virtual machine on which a new version of the system should be deployed for testing or even for production. Whereas, previously, system operators had to manually set up a server or configure a virtual machine when additional capacity was required or a new version of an operating system was available, infrastructure-as-code is used to automatically generate the right environment based on a specification developed in a dedicated programming language like `Puppet`, `Chef`, `CFEngine`, `Ansible` or `Salt`. For example, one could specify an Ubuntu 15.10 virtual machine with a specific version of the Apache web server installed, and even with a custom Apache configuration file in place.

Some infrastructure programming languages are declarative, while others rely on existing languages like Python or Ruby. All of them can interface either directly or through plugins with clouds, containers and virtual machines. Virtual machines are self-contained systems containing an operating system and every library or service required to emulate a server that can run the application under development. Since multiple virtual machines can execute on the same physical server, and can easily be turned on/off or migrated between servers, virtual machines form the backbone of modern cloud environments.

Containers are a lightweight alternative to virtual machines [23]. Since each virtual machine has its own version of the operating system, libraries and services, there is substantial duplication across all virtual machines, making them fairly inefficient and heavyweight. Instead, containers share as many components as possible in the form of "images," i.e., each container can be seen as a stack of images. Usually, most of the containers share the same operating system image and even images for web or application servers. Since images are read-only (to guarantee that they are not tampered with after creation), a container's specific state is typically stored in a custom image on top of its stack. Containers are said to save disk and memory space, and reduce run-time overhead [9].

Humble and Farley [36] recommend infrastructure code to be stored in a separate VCS repository than source code, in order to restrict access to infrastructure code. If not, anyone with regular source code access would be able to touch infrastructure code, enabling them to break the carefully configured testing or production environments (either by accident or on purpose) by adding potentially incompatible software or (in the worst case) injecting security vulnerabilities.

E. Deployment

Deployment is the phase in which the tested deliverables for the upcoming release are staged in preparation for release [9, 36]. For example, for web applications, deployment could correspond to pushing deliverables across a network to the correct directory on a web server, whereas, for mobile app deployment, to submitting an app's binary to the app store. The deployment phase is typically followed relatively quickly by the release phase (see below), since in between deployment and release, the deployed files basically are inactive (and invisible to users). One exception to this is the activity of "dark launching," which corresponds to deploying new features without releasing them to the public. The idea is that different parts of the system automatically make calls to the hidden features in a way invisible to end users (as the feature is hidden

for them). Dark launching is popular to test the scalability of a new feature with real work loads.

Different deployment (and release) strategies exist. "Blue/green deployment" deploys the next software version on a copy of the production environment (typically the test environment), then later on (during releasing), updates DNS or routing table entries to redirect incoming user traffic away from the old production environment to the new one. In "canary deployment," a prospective release of the software system is loaded onto a subset of the production environments (e.g., the servers in one country), only exposing a fraction of the user base to the new release. The performance and quality of the prospective release is closely monitored. If the "canary" is of sufficient quality, it is deployed on a larger segment of the production environments, and the release monitoring process resumes. This slow roll-out is continued until the whole production environment is running the new version. If issues are uncovered during the roll-out, the production environment can be quickly "rolled back" to the previously deployed release.

Finally, "A/B testing" deploys alternative A of a feature to the environment of a subset of the user base, while alternative B is deployed to the environment of another subset. Using release telemetry (see below), the alternatives are compared in order to decide which one is generating more interest from the user base. The more successful alternative is rolled out to the general user base.

F. Release

The final phase in a typical release engineering pipeline is to make deployed releases visible to system users [9]. When desktop software was still deployed to CD-ROMs, releasing corresponded to bringing the disks to shops for sale. Using the more modern deployment mechanisms for web apps listed above, releasing has become as simple as changing DNS entries or clicking a button to make a new mobile app release visible on the app store. Releasing mechanisms even allow fine-grained access to apps, as in the A/B testing case or when different system users may have access to different subsets of system functionality. For example, users who pay a premium may have access to a "deluxe" version of a system or early access to a release before it is made available to the public.

To make fine-grained enabling or disabling of a feature possible, for example to quickly disable a new feature of a web app that is causing substantial errors for customers, many companies again make use of feature toggles (see Section II-A). Just by changing the value of a feature toggle from "true" to "false" while the app is live, any future web request would no longer execute the now disabled feature. This feature toggling mechanism often is used in tandem with canary deployment [9]. Without feature toggles, rolling back a failing release is more complicated, especially when the database schema was modified for the new release. In those cases, many companies prefer "rolling forward," i.e., quickly making a follow-up release, either with the identified defects fixed or with the offending feature temporarily disabled.

Once a deployed version of a system is released, the release engineers monitor telemetry data and crash logs to track the performance and quality of releases. This is of course the easiest for web apps, but telemetry is also collected for desktop or even mobile apps. For web apps, a variety of off-the-shelf tools, such as `Nagios` and `Splunk`, can be used to collect and analyze release telemetry data. For mobile apps, third-party vendors offer custom frameworks, while for desktop apps organizations typically develop their own. These make organizations aware of the health of a release, and enables them to make data-driven decisions about release rollback, feature popularity, bug prevalence or feature scalability.

Finally, the release is also the moment for legal and administrative procedures such as sign-off for the official release (transferring management of the release to other teams for maintenance) and registration of the bill-of-materials [24], i.e., a detailed list of all files (and their revisions) going into the upcoming release. This bill-of-materials is essential for later maintenance activities and bug fixing, as one needs to know exactly what file revision to start analyzing and changing. An artifact repository (see Section II-B) enables automatic registration of a bill-of-materials.

III. Roadmap

This section presents our vision for research on release engineering, which has been inspired by the past three editions of the International Workshop on Release Engineering (RELENG), existing research on release engineering, and our prior research and practical experience in the field.

A. Integration: Branching and Merging

While a variety of branching structures have been proposed [6, 13, 19], no methodology or insight exists on how to empirically validate the best branching structure for a given organization or project. For example, Bird and Zimmermann [17] have proposed a means to evaluate the relative value of branches in an organization's branching structure. Although very promising, this work was only evaluated on one commercial organization's systems, and only considered the first step of "branch refactoring," i.e., the identification of suboptimal branching structures. Other identification criteria should be considered, as well as how to determine and physically refactor an existing branching structure.

Given a suitable branching structure, practitioners also require approaches and tools to predict the presence of integration conflicts [18, 32, 66, 89] and amount of effort required to resolve them. Release engineers need to pay particular attention to conflicts and incompatibilities caused by evolving library and API dependencies, where some components might migrate to a newer version of a library, while others stick to the existing version [57]. Being able to determine when it is safe (from the integration perspective) to update to a newer library or API version, or when the current dependency should be frozen, is a major area of future work. Above all, companies would like to know the optimal order in which teams should

merge back to minimize conflicts, or even how to cut a large merge into pieces that can be merged in an interleaved fashion.

Finally, given that the goal of the integration process (see Section II-A) is to bring high quality code changes as fast as possible to the master branch (and hence to the user), being able to profile the integration process for integration bottlenecks would be a major asset for companies. For example, companies would like to identify which branches see most of the merge conflicts, have the slowest code velocity or take the most time resolving conflicts [2]. Other branches might see much higher percentages of code changes that never make it to the master branch, because they fail integration tests or reviews somewhere along the branch structure. Methodologies based on mining of version control and defect repositories could build models and tools to pinpoint the bottlenecks in the integration phase [30], while qualitative analyses like interviews could help to understand the cause of such bottlenecks.

B. Continuous Integration: Building and Testing

Given that CI servers need to initiate build and test jobs for each code change committed to a project's VCS, across all branches, and each job should be repeated across all supported platforms and multiple build configurations, speeding up CI might be the major concern of practitioners. Apart from optimizing the tests (outside the scope of this paper, see for example Herzig et al. [35] and Yoo and Harman [91]) and build specifications (see below), the main focus is to reduce the number of CI jobs that should be started. Different strategies can be explored for this.

A first strategy is to predict whether a code change will break the build. For example, based on build results of similar code changes, one could build classification models that can say with enough certainty that a new code change cannot compile or will fail the test suite. In that case, instead of starting compilation or testing, one can immediately contact the developer or reviewers to take a close look and fix any glaring problems. Some existing prediction models exist [34, 41], but none of them are able to perform prediction across platforms or configurations. Indeed, a code change might build and test cleanly on Windows 10, but could fail on OSX, or might only fail when feature A and B are built together.

A second strategy is to "chunk" code changes into a group and only compile and test each group once. If the build and test pass, one is sure that none of the commits had a problem. However, if the chunk-level build fails, it is not clear which of the commits in the chunk is responsible for the breakage, and one would need to rebuild each commit in the chunk individually to identify the culprit(s). Projects like OpenStack, where the number of commits per day is extremely high, have started experimenting with more clever chunking approaches that can find the commit(s) responsible for a build breakage faster. However, thorough empirical evaluation and improvement of such approaches currently is missing.

If all else fails, using more powerful servers, network switches and hard disks is an approach to make heavier build loads feasible [2]. Unfortunately, the load of the CI system fluctuates across time, for example depending on the kind of features being developed or the progress of the current development sprint. Hence, practitioners would like to predict the expected build load and instantiate virtual machines accordingly, possibly in a cloud environment. Such predictions need data about previous code changes, build load and build failures in order to build statistical or data mining models.

There are other challenges regarding CI as well. For one, the concept of "green builds" slowly is becoming an issue, in the sense that frequent triggering of the CI server consumes energy. Apart from trying to speed up builds, or automatically reduce some part of the CI server's energy consumption, one could also try reducing network communication and/or disk access to reduce energy consumption of the build infrastructure and hence the energy bill for an organization's server farm.

Lately, security of the release engineering pipeline in general, and the CI server in particular, also has become a major concern [9], since any malicious script capable of injecting a payload in the CI's build output could infect the rest of the release engineering pipeline. Indeed, the build results of CI are stored in the artifact repository, then reused *as is* in all subsequent pipeline phases. Whereas, for a given code change, this ensures usage of identical binaries throughout, it also means that problematic deliverables having defects or malware can pass as-is. Research should investigate the implications of this security problem and what can be done about it.

C. Build System

Build correctness and build performance have been a focal point of much prior work. A build system is said to be correct if, given a set of source code files, it generates the deliverables desired by the developer. On the other hand, a build is said to achieve high performance if re-executing the build after a code change only requires a minimum number of build operations to be performed. Improvements to build performance are often made at the expense of build correctness, and vice versa [3]. Below, we discuss both challenges in more detail.

Build correctness is challenging to measure or even observe, unless a build fails with explicit error messages. In the worst case, an incorrect build system may finish successfully, but generate binaries that have unexpected test or execution problems [48]. Existing work has used both dynamic [4, 83] and static [21, 53, 79] approaches to check correctness of the two layers of a build system, but suffers from the same bias towards classic build technologies like GNU Make or Apache Ant as CI research is experiencing. New build technologies like `Gradle` or Google's `Bazel` simplify assumptions made by older technology, which may have an impact on build correctness. Recently, a new kind of build correctness, i.e., "reproducible builds", has surfaced among software organizations [28, 61, 67], according to which, for a given feature and hardware configuration of the code base, every build invocation should yield bit-to-bit identical build results. Reproducible builds are a prerequisite to reliably identify security breaches or platform-specific problems on a build machine.

In general, the correctness of a build system is impacted by the way in which build specifications co-evolve with the source code, for example to include new source code files in the build. The evolution and maintenance of build system specification files primarily has been studied quantitatively [33, 48, 77]. With the exception of Shridhar et al. [72], qualitative analysis of build system evolution and maintenance is largely missing. Qualitative studies are not only essential to understand the rationale behind quantitative findings (e.g., why did an organization decide to migrate to another build technology?), but also to identify design patterns and best practices for build systems. How can developers make their builds more maintainable and of higher quality? What refactorings should be performed for which build system anti-patterns? Whereas the specifics of these refactorings likely are technology-specific, the larger principles could apply to specific families of build system technologies (e.g., the task-based ones).

Another largely untouched area of build correctness is the identification and resolution of build bugs, i.e., source code or build specification changes that cause build breakage, possibly on a subset of the supported platforms. Similar to source code, defect prediction models could be built, or other heuristics that leverage static, dynamic or historical build data to identify broken build specifications ahead of time. Furthermore, while basic debuggers for GNU Make [14, 31] and Apache Ant [56, 78] exist, advanced debugging tools, especially for more modern build technologies, are still missing.

Build performance, on the other hand, refers to the challenge of making a personal or CI build as fast as possible. Although all build system tools provide incremental build functionality that only rebuilds what is necessary after a code change, basic tools have a hard time determining what is necessary. For this reason, many projects use solutions for building in the cloud or for advanced build caching like Electric Cloud's `ElectricAccelerator` and `ccache`. Various papers have focused on speeding up C/C++ build systems [20, 47], especially by optimizing dependencies on header files (since any change to a header file potentially impacts the compilation of dozens of other files).

However, what is missing are studies on non-GNU Make build systems. Such build systems are not file-based, but task- or workflow-based, and hence use a higher level abstraction than header files. Apart from identifying bottlenecks, such approaches should also suggest concrete refactorings of the build system specifications [20] or source code [84].

D. Infrastructure-as-Code

Given that infrastructure-as-code, together with modern deployment technology like containers, is one of the more recent components of modern release engineering pipelines, there are many open avenues for research. Current work [37] focuses on maintenance of infrastructure-as-code, inspired by how research on build systems began. Infrastructure files can be large and change drastically relative to their size, which suggests an important potential for bug-proneness. Hence, the link between infrastructure-as-code and software quality needs to be established, as well as any hidden co-evolution relation with (for example) source code and test files.

We expect the next wave of research on infrastructure-as-code to focus on the differences between infrastructure languages, since a language like Puppet uses a different paradigm than Chef. Hence, maintenance effort as well as quality measures will likely be different between these languages. Furthermore, in discussions with companies, it has become clear that, again similar to build system technology [77], companies migrate from one infrastructure technology to another if they are unhappy with the development or maintenance of these files. Until sufficient VCS data becomes available for infrastructure-as-code in open source and industrial projects, qualitative analyses involving developers will be required.

Above all, developers are looking for best practices and design patterns for infrastructure-as-code. Indeed, similar to regular programming languages, languages like Puppet offer various ways to obtain the same result, but some require more maintenance or result in more complex files. Qualitative analysis of infrastructure code will be necessary to understand how developers address different infrastructure needs. Quantitative analysis of the version control and bug report systems can then help to determine which patterns were beneficial in terms of maintenance effort and/or quality.

E. Deployment

Continuous deployment and delivery are concepts that have been known and used for at least 5 years [36]. For example, Facebook's Chuck Rossi ("solved problem") [65] and former Etsy's Noah Sussman ("continuous delivery is mainstream") [76] indicated that for large IT companies, implementing continuous delivery for web apps is straightforward, given that tools have become widely available or even commodities. Even desktop applications like browsers (Google Chrome and Mozilla Firefox) have adopted regular, rapid release schedules for several years.

Hence, the major deployment challenge in the web and desktop areas has shifted towards distilling essential and/or best practices for the smaller software companies and start-ups, by obtaining empirical evidence of what works and what fails. While large companies with substantial resources can afford trial-and-error and even failure (as discussed in the various keynote talks at the RELENG workshop [1]), younger companies with smaller margins for error do not have this luxury and want to adopt the right tools and practices from the start. For example, is blue-green deployment the fastest means to deploy a new version of a web app? Are A/B testing and dark launching worth the investment and risk (e.g., in terms of security and privacy)? Should one use containers or virtual machines for a medium-sized web app in order to meet application performance and robustness criteria? If an app is part of a suite of apps built around a common database, should each app be deployed in a different container?

Apart from informing smaller companies, the other remaining deployment challenge in the era of continuous delivery is the interaction with non-traditional deployment environments.

The most common example right now are mobile app stores. Mobile apps are a huge market, with 1.5 and 1.6 million mobile apps respectively in the Apple and Google Play App Stores in July 2015 [75], with the Apple App Store alone responsible for 100 billion app downloads (cumulative across time for free and commercial apps together) by June 2015 [74]. While web apps can be used via a browser, many web app companies develop their own native or HTML5 apps for the major mobile platforms. These mobile apps need to stay in sync with the evolution of the web apps. Ideally, new features should be released on all platforms at once (if it makes sense to port them to the smaller screen estate of a mobile device).

While, planning-wise, releasing a product across different platforms is already a challenge, there is the extra complication of inversion-of-deployment-control. While many web and desktop app companies try to control deployment (and release) of a new version of their app by hosting their own web app or building an automated update mechanism into their desktop product, current mobile app platform vendors do not allow this. Indeed, deployment and release currently is controlled by the platform's app store (e.g., Google Play or Apple App Store), which is a catalog of all mobile apps offered on a platform, with a marketing front-end through which customers can seamlessly buy and/or download the apps.

Once a company has uploaded its new release of an app, the app store performs vetting and review activities, after which, depending on the queue of new app releases lining up, the app will be deployed and released. This makes it hard to accurately project a release date, especially if an app release accidentally violates some app store policies and needs to be fixed, re-uploaded and re-vetted. Even worse is the case of showstopper bugs pointed out by users right after release. For a web app or even a desktop app with an update mechanism, as soon as the bug is fixed, the fix can be deployed and released. For a mobile app, the fix needs to pass again through app store vetting process. Although a special queue exists for such cases, the speed of deployment again depends on external factors, during which time users may have already posted negative reviews or ratings on an app's app store page, which may deter other mobile users from buying the app [42].

Although inversion-of-deployment-control is partly an institutional problem, researchers can help as well. For one, better tools for quality assurance are required, to prevent showstopper bugs from slipping through and requiring re-deployment of a mobile app version (with corresponding vetting). These could range from defect prediction (either file- or commit-based), to smarter/safer update mechanisms, tools for improving code review, generating tests, filtering and interpreting crash reports or even prioritization and triaging of defect reports. Granted, not all of these challenges are release engineering-specific, but these activities would definitely reduce the risk of a bad mobile release for developers.

The owner of a mobile platform, on the other hand, needs approaches to quickly determine the changes to a mobile app release and review them (for example in case of an emergency fix to a recent release), or at least shorten the time from bug detection to deployment of a fixed version. One recent proposal is for app stores to generate temporary fixes for certain kinds of defects, while the app developers are working on a permanent fix [29]. In the absence of more deploy and release autonomy for app companies, such techniques could at least reduce the pain for app release engineering teams.

F. Release

Once a candidate release has arrived at the final phase of the release engineering pipeline, i.e., the actual release of a newly deployed version, there are still release finalization issues that need to be addressed. For example, for quality assurance purposes, companies do not only make a final release, but make intermediate alpha and beta releases, and after the final releases might need to make minor or patch releases to iron out reported defects or bring small feature improvements to users. Even the final release might not be released all at once, but might be revealed (released) to one user group at a time (see canary deployment in Section II-E). Finally, companies like Netflix promote a "roll your own release" (RYOR) policy, where development teams are given the tools to manage releases themselves. In all these cases, the challenge is to determine which code change is the perfect one for triggering the release of one of these releases, or whether a canary is good enough to be released to another data centre (or whether it should be rolled back from all data centres on which it is currently released).

For web apps, this decision currently is based on any feedback from the field through deployment or release logs, crash reports, mobile app reviews or ratings [9]. Logs and reports typically contain stack traces or error messages that are either interpreted automatically or manually, while reviews contain users' praise or critiques of a new release. As soon as an abnormal peak in errors is observed, the release engineers (typically supported by devops and software engineers) need to make a trade-off between leaving the faulty release in place (waiting for the next one to resolve defects), rolling it back (reverting it) by going back to the previous release or rolling forward by quickly releasing a fix. Helping stakeholders make this important trade-off, for example by balancing costs and benefits of a rollback compared to a roll-forward, as well as helping to prioritize field problems right after a release, are major challenges [40].

For desktop and mobile apps, similar feedback mechanisms exist. For example, third-party services offer crash report and analytics frameworks that collect data and notify teams of field problems. Still, similar to web apps, decision making is left up to the developers. Desktop and mobile apps have the additional challenge of having to cover multiple platforms. Hence, should one release on all platforms at the same time? In the case of defects, which platform should receive priority? Should all platforms use the same version numbering, or should that be feature-dependent, with different platforms potentially having different features [58]?

Finally, recent studies have started to evaluate the concept of continuous delivery and rapid releases from the perspective

of software quality. Studies on Google Chrome [10] and Mozilla Firefox [43, 45] found initial benefits and costs of these concepts, however studies on other systems (e.g., mobile apps [55]) as well as from other perspectives should be performed in order to help companies decide whether changing release engineering strategy would be necessary for them. For example, companies like to use the metaphor of "release trains" to explain that since the next release (train) is only a short time away, developers are no longer frustrated when missing a release. However, such psychological effects have not been validated.

IV. A Release Engineering Checklist for SE Researchers

Based on our experience, this section provides concrete advice about how release engineering can impact software engineering researchers who need to mine software repositories. We present this advice in the form of a checklist, setting the stage with a fictional conversation between a naive professor and clever student, then discussing what is the recommended course of action in order to preserve the integrity of the research findings.

A. Not All Releases are Equal

> *Prof:* Look at this project! It had a crazy amount of post-release bugs within 6 months after release!
> *Student:* Well, it releases once every 6 weeks, so you're basically counting the bugs of 4 releases ...

Every project has its own release schedule, with some projects following a time-based schedule (e.g., weekly or monthly releases), others a feature-based schedule ("as soon as this is done") and others not following any schedule at all. As a result, for one project 4 weeks might be negligible compared to the total cycle time, while for another project this might be the full cycle time between two subsequent releases. In other words, one cannot compare projects with different release schedules head-to-head in terms of an absolute number of weeks or months.

On the other hand, even with short cycle times, developers have typically been working on a particular feature for a much longer time than one release cycle. For example, by the time a version of Mozilla Firefox is released, it has been under nightly/alpha/beta testing for 18 weeks, and (before that) under development for several more weeks! Such projects use a staged release pipeline (akin to superscalar processor architectures [80]), where release i is in beta, while release $i-1$ is in alpha and $i-2$ in nightly testing. Hence, releases have a longer development tail than their cycle time would suggest.

Related to this is the practice of feature freezing X weeks before a release, at which point no new features will be accepted for the upcoming release and stabilization of the upcoming release is performed [60]. For example, release engineers make a release branch, while ongoing development of new features continues on the master branch. If a study by accident would be measuring the bugs reported during feature freeze or the code changes reviewed in that period, a much larger proportion of bug fixes would be visible than usual, with hardly any new feature development [27].

Other factors that can impact the findings of an empirical study are the differences in scope between major, minor and patch releases, as well as the interleaving of releases. The latter refers to how releases with successive major version numbers are situated relative to each other. For example, if a project releases version 1.0, 1.1 and 1.2, before moving to 2.0 and 2.1, all releases are sequential, with only one release considered to be the newest one. Hence, any bugs reported in between the release of 1.2 and 2.0 likely refer to 1.2 (or maybe a pre-release version of 2.0). In contrast, if the 1.x series is being maintained while the 2.x series is released, the chronological order of releases could be 1.0, 1.1, 2.0, 1.2 and 2.1. In that case, different release series' are interleaved, and any bug reports added between 2.0 and 1.2 could apply to both 1.1 or 2.0.

The best way to address these release-related problems is to first study the release schedule followed by a project. In particular, researchers should understand the time intervals between releases, the nature of the releases (major, minor or patch) and the interaction between releases (sequential vs. interleaved). Depending on the study, minor releases might not be as relevant as major releases. Hence, the data might need to be filtered based on the types of releases of interest. Furthermore, any repository should be checked for explicit links to releases. If bugs are tagged by the release they were reported in, a study will be safe even for interleaved releases.

Similarly, if the actual development period of a release is important, the study should trace back each release to the branches the work originated from (see below). This avoids limiting the study only to commits made during the cycle time, or accidentally capturing the stabilization period instead of actual feature development. Finally, one should check multiple sources to obtain the correct release dates. Version control commits tagged with a specific release do not necessarily correspond to the actual release date, since the integration phase is distinct from the deployment and release phases (Section II), and hence a variety of activities might need to take place after the final commit of a release.

B. There are Branches, and Branches

> *Prof:* Weird, the size of this project keeps on fluctuating between 50,000 and 45,000 lines ...
> *Student:* Of course, did you filter out the re-engineering branch active across all of 2014?

Most software projects have more than one branch open at a time, especially given the popularity of distributed version control systems. As discussed in Section II-A, in such projects code flow is modeled as branches being merged into another branch, reflected by a so-called merge commit that connects the branches together. Apart from expressing that the commits on the child branch are now part of the parent branch, merge commits also contain the changes required to resolve any conflicts between the branches. Hence, for many types of

analyses, one might need to ignore merge commits, and only focus on the actual commits on the branches.

Apart from eliminating merge commits, one also needs to select which branches are of interest, i.e., one, multiple or all. If a unique order is required across the (non-merge) commits to be analyzed, one will typically need to project all commits onto one branch. This is because one cannot blindly trust commit dates to track the evolution of a metric across commits developed in different branches (as they were made in parallel). This means that one should study (1) the non-merge commits made directly to that branch and (2) replace the merge commits by a virtual commit corresponding to the total set of code changes merged into the parent branch. A VCS like git provides commands for this.

Hence, if we assume that one wants to track the evolution across commits of a particular metric, one should first select the branch of interest and identify its merge commits. Then, one should calculate the metric of interest for each non-merge commit of each branch. Finally, for merge commits, one should aggregate the metric values across all commits of the branch that it is merging. For example, to measure the evolution of code churn (i.e., size of a commit), one should first calculate the churn for each non-merge commit (counting the number of changed lines of code). The churn of a merge commit should then be calculated as the total churn merged into the branch of interest. Finally, one can plot the chronological evolution of the resulting churn values according to the (non-)merge commit dates.

Finally, researchers should take special note of a project's release branch, which is a branch typically created when a release goes into feature-freeze to host all of its (bug) stabilization work. While this stabilization is being done, regular development for a future release continues on the master and other branches, which means that regular commits chronologically are interleaved with the bug fixing commits on the release branch. Similar care as above should be taken to not mix up both kinds of commits, as well as to determine which commits end up in a particular release. The commit date by itself does not suffice to link commits to releases.

C. Choose before you Build

> *Prof:* This browser has a huge code base, it even has code for an email client, calendar and chat support!
> *Student:* Yes, but did you notice that these are 4 systems sharing the same code base, with the build system deciding which one is being built?

Not every file in a project's repository goes into a release [85]. Similar to software product lines, most projects can be configured at compile-time (e.g., using the build configuration) and sometimes even at run-time (e.g., using run-time feature toggles) to select which features should be compiled into a binary (or executed at run-time), and what platforms should be targeted. For example, the Linux kernel can be configured to include popular drivers or only drivers that are needed for the target machine.

Selection of features typically implies that only the files corresponding to those features, or containing at least some code of those features, will be considered for compilation. Hence, dynamic analysis of the corresponding binary would only be able to traverse code in those files, and would be unaware of any of the ignored source code files. Conversely, static analysis might conclude that the system is larger and more complex than it actually is. For example, the Linux kernel has thousands of device drivers, but most deployed kernels use only a small fraction of them. If a project uses run-time feature toggles, some of the code files that are included in the resulting binary would be disabled, unless the feature configuration is changed.

Platform configuration corresponds to deciding between the target operating system (e.g., Windows vs. Linux vs. OSX), supported hardware or even what versions of libraries to select. Again, certain parts of a project's repository will only be included in a binary if the right operating system or hardware is selected. For example, projects typically have specific directories for Windows-only implementations of some code files, while other directories have the corresponding implementations for other operating systems. No compiled version will include the files of both directories.

Finally, note that the choice of platform (e.g., a library version) also provides a limitation in the sense that to analyze a project at some point in time one will need to use exactly the same versions of libraries and tools. For example, our earlier work on the evolution of the Linux kernel build system forced us to install legacy gcc versions (2.95 and older) to compile old kernel versions, since those featured compiler-specific (and machine-specific) assembler dialects [3, 77].

V. Conclusions

Release engineers develop, maintain and operate the pipelines that deliver the new features and bug fixes produced by developers to the end user in the form of high quality software releases. The importance of release pipelines in capturing and retaining end users has lead to many recent advances from industry. Yet, researchers can play an essential role in addressing the plethora of open questions that hinder the release engineers of modern software organizations.

In this paper, we set out to equip software engineering researchers with an understanding of release engineering and its impact on their own research. We first provide researchers with a working definition of modern release engineering pipelines and their data repositories. Moreover, we discuss each release engineering activity, proposing promising avenues for research. Finally, we provide a checklist of release engineering decisions that, if not carefully considered, can lead to incorrect conclusions when mining software repository data. Indeed, if software engineering researchers would need to remember one section of this paper, the checklist would be the one, even if their research does not specifically target the release engineering process!

ACKNOWLEDGMENTS

We would like to thank the vibrant RELENG community for its openness and wealth of insights.

REFERENCES

[1] B. Adams, S. Bellomo, C. Bird, F. Khomh, and K. Moir, "Intl. workshop on release engineering (RELENG)," http://releng.polymtl.ca.

[2] B. Adams, S. Bellomo, C. Bird, T. Marshall-Keim, F. Khomh, and K. Moir, "The practice and future of release engineering - a roundtable with three release engineers," *IEEE Software*, vol. 32, no. 2, pp. 42–49, 2015.

[3] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter, "The evolution of the linux build system," *Electronic Communications of the ECEASST*, vol. 8, February 2008.

[4] B. Adams, K. De Schutter, H. Tromp, and W. D. Meuter, "Design recovery and maintenance of build systems," in *Proc. of the Intl. Conf. on Soft. Maint.*, 2007, pp. 114–123.

[5] B. Adams, R. Kavanagh, A. E. Hassan, and D. M. German, "An empirical study of integration activities in distributions of open source software," *Empirical Software Engineering*, 2015, to appear.

[6] R. Aiello and L. Sachs, *Configuration Management Best Practices: Practical Methods That Work in the Real World*, 1st ed. Addison-Wesley Professional, 2010.

[7] J. M. Al-Kofahi, H. V. Nguyen, A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Detecting semantic changes in makefile build code," in *Proc. of the Intl. Conf. on Software Maintenance (ICSM)*, 2012, pp. 150–159.

[8] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proc. of the Intl. Conf. on Software Engineering (ICSE)*, 2013, pp. 712–721.

[9] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*, 1st ed., ser. SEI Series in Software Engineering. Addison-Wesley Professional, May 2015.

[10] O. Baysal, I. Davis, and M. W. Godfrey, "A tale of two browsers," in *Proc. of the 8th Working Conf. on Mining Software Repositories (MSR)*, 2011, pp. 238–241.

[11] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "The influence of non-technical factors on code review," in *Proc. of the Working Conf. on Reverse Engineering (WCRE)*, 2013, pp. 122–131.

[12] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proc. of the Working Conf. on Mining Software Repositories (MSR)*, 2014, pp. 202–211.

[13] S. P. Berczuk and B. Appleton, *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[14] R. Bernstein, "Debugging makefiles with remake," in *Proc. of the 25th Intl. Conf. on Large Installation System Administration (LISA)*, 2011, pp. 27–27.

[15] N. Bettenburg, A. E. Hassan, B. Adams, and D. M. German, "Management of community contributions: A case study on the android and Linux software ecosystems," *Empirical Software Engineering*, vol. 20, no. 1, pp. 252–289, February 2015.

[16] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *Proc. of the 2009 6th IEEE Intl. Working Conf. on Mining Software Repositories (MSR)*, 2009, pp. 1–10.

[17] C. Bird and T. Zimmermann, "Assessing the value of branches with what-if analysis," in *Proc. of the Intl. Symp. on Foundations of Software Engineering (FSE)*, 2012.

[18] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proc. of the 19th ACM SIGSOFT Symp. and the 13th European Conf. on Foundations of Software Engineering (ESEC/FSE)*, 2011, pp. 168–178.

[19] S. Chacon, *Pro Git*, 1st ed. Berkely, CA, USA: Apress, 2009.

[20] H. Dayani-Fard, Y. Yu, J. Mylopoulos, and P. Andritsos, "Improving the build architecture of legacy c/c++ software systems," in *Proc. of the 8th Intl. Conf., Held As Part of the Joint European Conf. on Theory and Practice of Software Conf. on Fundamental Approaches to Software Engineering (FASE)*, 2005, pp. 96–110.

[21] M. de Jonge, "Build-level components," *IEEE Trans. Softw. Eng.*, vol. 31, no. 7, pp. 588–600, Jul. 2005.

[22] A. Dearle, "Software deployment, past, present and future," in *2007 Future of Software Engineering (FOSE)*, 2007, pp. 269–284.

[23] I. Docker, "Docker," https://www.docker.com.

[24] E. Dolstra, M. de Jonge, and E. Visser, "Nix: A safe and policy-free system for software deployment," in *Proc. of the 18th USENIX conf. on System admin.*, 2004, pp. 79–92.

[25] P. Duvall, S. M. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007.

[26] T. Fitz, "Continuous deployment at IMVU: Doing the impossible fifty times a day," http://timothyfitz.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/, February 2009.

[27] D. M. German, B. Adams, and A. E. Hassan, "Continuously mining the use of distributed version control systems: An empirical study of how linux uses git," *Empirical Software Engineering*, 2015, to appear.

[28] "Gitian," https://gitian.org.

[29] M. Gómez, M. Martinez, M. Monperrus, and R. Rouvoy, "When app stores listen to the crowd to fight bugs in the wild," in *Proc. of the 37th Intl. Conf. on Software Engineering (ICSE) - Volume 2*, 2015, pp. 567–570.

[30] G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *Proc. of the Intl. Conf. on Software Engineering (ICSE)*, 2015, pp. 358–368.

[31] J. Graham-Cumming, "Gnu make debugger," http://gmd.sourceforge.net, 2014.

[32] M. L. Guimarães and A. R. Silva, "Improving early detection of software merge conflicts," in *Proc. of the 34th Intl. Conf. on Software Engineering (ICSE)*, 2012, pp. 342–352.

[33] R. Hardt and E. V. Munson, "An empirical evaluation of ant build maintenance using formiga," in *Proc. of the Intl. Conf. on Software Maintenance and Evolution*, 2015, pp. 201–210.

[34] A. E. Hassan and K. Zhang, "Using decision trees to predict the certification result of a build," in *Proc. of the 21st IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE)*, 2006, pp. 189–198.

[35] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *Proc. of the 37th Intl. Conf. on Software Engineering (ICSE) - Volume 1*, 2015, pp. 483–493.

[36] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.

[37] Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code: An empirical study," in *Proc. of the 12th Working Conf. on Mining Software Repositories (MSR)*, 2015, pp. 45–55.

[38] Y. Jiang, B. Adams, and D. M. German, "Will my patch make it? and how fast? – case study on the linux kernel," in *Proc. of the 10th IEEE Working Conf. on Mining Software Repositories (MSR)*, 2013, pp. 101–110.

[39] C. F. Kemerer and M. C. Paulk, "The impact of design and code reviews on software quality: An empirical study based on psp data," *Transactions on Software Engineering (TSE)*, vol. 35, no. 4, pp. 534–550, April 2009.

[40] N. Kerzazi and B. Adams, "Botched releases: Do we need to roll back? empirical study on a commercial web app," in *Proc. of the 23rd IEEE Intl. Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, Osaka, Japan, March 2016.

[41] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," in *Proc. of the 30th IEEE Intl. Conf. on Software Maintenance (ICSM)*, 2014, pp. 41–50.

[42] H. Khalid, E. Shihab, M. Nagappan, and A. Hassan, "What do mobile app users complain about?" *Software, IEEE*, vol. 32, no. 3, pp. 70–77, May 2015.

[43] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality? an empirical case study of mozilla firefox,"

in *Proc. of the Working Conf. on Mining Software Repositories (MSR)*, 2012, pp. 179–188.

[44] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?" in *Proc. of the Intl. Conf. on Software Maintenance and Evolution (ICSME)*, 2015, pp. 111–120.

[45] M. Mäntylä, F. Khomh, B. Adams, E. Engström, and K. Petersen, "On rapid releases and software testing," in *Proc. of the 29th IEEE Intl. Conf. on Software Maintenance (ICSM)*, 2013, pp. 20–29.

[46] M. V. Mäntylä and C. Lassenius, "What types of defects are really discovered in code reviews?" *Transactions on Software Engineering (TSE)*, vol. 35, no. 3, pp. 430–448, June 2009.

[47] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan, "Identifying and understanding header file hotspots in c/c++ build processes," *Automated Software Engineering*, 2015, to appear.

[48] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proc. of the 33rd Intl. Conf. on Software Engineering (ICSE)*, 2011, pp. 141–150.

[49] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. In press, 2015.

[50] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan, "A large-scale empirical study of the relationship between build technology and build maintenance," *Empirical Software Engineering*, vol. In press, 2015.

[51] A. Meneely, A. C. R. Tejeda, B. Spates, S. Trudeau, D. Neuberger, K. Whitlock, C. Ketant, and K. Davis, "An empirical investigation of socio-technical code review metrics and security vulnerabilities," in *Proc. of the Intl. Workshop on Social Software Engineering (SSE)*, 2014, pp. 37–44.

[52] R. Morales, S. McIntosh, and F. Khomh, "Do code review practices impact design quality? a case study of the qt, vtk, and itk projects," in *Proc. of the Intl. Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 171–180.

[53] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: Static analyses and empirical results," in *Proc. of the Intl. Conf. on Software Engineering (ICSE)*, 2014, pp. 140–151.

[54] S. Nadi, C. Dietrich, R. Tartler, R. C. Holt, and D. Lohmann, "Linux variability anomalies: What causes them and how do they get fixed?" in *Proc. of the Working Conf. on Mining Software Repositories (MSR)*, 2013, pp. 111–120.

[55] M. Nayebi, B. Adams, and G. Ruhe, "Release practices in mobile apps - users and developers perception," in *Proc. of the 23rd IEEE Intl. Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[56] A. project, "Antelope," http://antelope.stage.tigris.org, 2009.

[57] S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *Proc. of the 28th IEEE Intl. Conf. on Software Maintenance (ICSM)*, 2012, pp. 378–387.

[58] ——, "Semantic versioning versus breaking changes: a study of the maven repository," in *Proc. of the 14th Intl. Working Conf. on Source Code Analysis and Manipulation (SCAM)*, 2014, pp. 215–224.

[59] A. Rahman, E. Helms, L. Williams, and C. Parnin, "Synthesizing continuous deployment practices used in software development," in *Proc. of the 2015 Agile Conf. (AGILE)*, 2015, pp. 1–10.

[60] M. Rahman and P. Rigby, "Release stabilization on linux and chrome," *Software, IEEE*, vol. 32, no. 2, pp. 81–88, Mar 2015.

[61] "Provide a verifiable path from source code to binary," https://reproducible-builds.org.

[62] P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices: A case study of the apache server," in *Proc. of the Intl. Conf. on Software Engineering (ICSE)*, 2008, pp. 541–550.

[63] P. C. Rigby and M.-A. Storey, "Understanding broadcast based peer review on open source software projects," in *Proc. of the Intl. Conf. on Software Engineering (ICSE)*, 2011, pp. 541–550.

[64] P. Rodríguez, A. Haghighatkhah, L. E. Lwakatare, S. Teppola, T. Suomalainen, J. Eskeli, T. Karvonen, P. Kuvaja, J. M. Verner, and M. Oivo, "Continuous deployment of software intensive products and services: A systematic mapping study," *Journal of Systems and Software (JSS)*, 2016, to appear.

[65] C. Rossi, "Moving to mobile: The challenges of moving from web to mobile releases," Keynote at RELENG 2014, April 2014, https://www.youtube.com/watch?v=Nffzkkdq7GM.

[66] A. Sarma, Z. Noroozi, and A. van der Hoek, "Palantir: raising awareness among configuration management workspaces," in *Proc. of the 25th Intl. Conf. on Software Engineering (ICSE)*, 2003, pp. 444–454.

[67] S. Schoen and M. Perry, "Why and how of reproducible builds: Distrusting our own infrastructure for safer software releases," https://air.mozilla.org/why-and-how-of-reproducible-builds-distrusting-our-own-infrastructure\-for-safer-software-releases/, November 2014.

[68] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at google)," in *Proc. of the Intl. Conf. on Software Engineering (ICSE)*, 2014, pp. 724–734.

[69] S. Shankland, "Google ethos speeds up chrome release cycle," http://cnet.co/wlS24U, July 2010.

[70] ——, "Rapid-release firefox meets corporate backlash," http://cnet.co/ktBsUU, June 2011.

[71] E. Shihab, C. Bird, and T. Zimmermann, "The effect of branching strategies on software quality," in *Proc. of the Intl. Symp. on Empirical Software Engineering and Measurement (ESEM)*, 2012, pp. 301–310.

[72] M. Shridhar, B. Adams, and F. Khomh, "A qualitative analysis of software build system changes and build ownership styles," in *Proc. of the 8th Intl. Symp. on Empirical Software Engineering and Measurement (ESEM)*, 2014.

[73] D. Ståhl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *J. Syst. Softw.*, vol. 87, pp. 48–59, Jan. 2014.

[74] Statista, "Cumulative number of apps downloaded from the apple app store from july 2008 to june 2015 (in billions)," http://www.statista.com/statistics/263794/number-of-downloads-from-the-apple-app-store/, June 2015.

[75] ——, "Number of apps available in leading app stores as of july 2015," http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, July 2015.

[76] N. Sussman, "Continuous delivery is mainstream," http://infiniteundo.com/post/71540519157/continuous-delivery-is-mainstream, 2014.

[77] R. Suvorov, B. Adams, M. Nagappan, A. Hassan, and Y. Zou, "An empirical study of build system migrations in practice: Case studies on kde and the linux kernel," in *Proc. of the 28th IEEE Intl. Conf. on Software Maintenance (ICSM)*, 2012, pp. 160–169.

[78] P. Systems, "Virtual ant," http://www.placidsystems.com/virtualant/, 2007.

[79] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, "Build code analysis with symbolic evaluation," in *Proc. of the Intl. Conf. on Software Engineering (ICSE)*, 2012, pp. 650–660.

[80] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Prentice Hall Press, 2014.

[81] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Investigating code review practices in defective files: An empirical study of the qt system," in *Proc. of the Working Conf. on Mining Software Repositories (MSR)*, 2015, pp. 168–179.

[82] "Mozilla try server," https://wiki.mozilla.org/ReleaseEngineering/TryServer.

[83] Q. Tu and M. W. Godfrey, "The build-time software architecture view," in *Proc. of the IEEE Intl. Conf. on Software Maintenance (ICSM)*, 2001, pp. 398–407.

[84] M. Vakilian, R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni, "Automated decomposition of build targets," in *Proc. of the 37th Intl. Conf. on Software Engineering (ICSE) - Volume 1*, 2015, pp. 123–133.

[85] S. van der Burg, E. Dolstra, S. McIntosh, J. Davies, D. M. German, and A. Hemel, "Tracing software build processes to uncover license compliance inconsistencies," in *Proc. of the 29th ACM/IEEE Intl. Conf. on Automated Software Engineering (ASE)*, 2014, pp. 731–742.

[86] A. van der Hoek and A. L. Wolf, "Software release management for component-based software," *Softw. Pract. Exper.*, vol. 33, no. 1, pp. 77–

98, Jan. 2003.

[87] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *Proc. of the Joint Meeting of the European Software Engineering Conf. and the Symp. on the Foundations of Software Engineering (ESEC/FSE)*, 2015, pp. 805–816.

[88] J. A. Whittaker, J. Arbon, and J. Carollo, *How Google Tests Software*, 1st ed. Addison-Wesley Professional, 2012.

[89] J. Wloka, B. Ryder, F. Tip, and X. Ren, "Safe-commit analysis to facilitate team software development," in *Proc. of the 31st Intl. Conf. on Software Engineering (ICSE)*, 2009, pp. 507–517.

[90] XebiaLabs, "Periodic table of devops tools," https://xebialabs.com/periodic-table-of-devops-tools/, 2015.

[91] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012.