2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

Feature Toggles: Practitioner Practices and a Case Study

Md Tajmilur Rahman†, Louis-Philippe Querel†, Peter C. Rigby†, Bram Adams§
† Concordia University, § Polytechnique Montreal
Montreal, QC, Canada
{mdt_rahm, l_querel}@encs.concordia.ca
peter.rigby@concordia.ca, bram.adams@polymtl.ca

ABSTRACT

Continuous delivery and rapid releases have led to innovative techniques for integrating new features and bug fixes into a new release faster. To reduce the probability of integration conflicts, major software companies, including Google, Facebook and Netflix, use *feature toggles* to incrementally integrate and test new features instead of integrating the feature only when it's ready. Even after release, feature toggles allow operations managers to quickly disable a new feature that is behaving erratically or to enable certain features only for certain groups of customers. Since literature on feature toggles is surprisingly slim, this paper tries to understand the prevalence and impact of feature toggles. First, we conducted a quantitative analysis of feature toggle usage across 39 releases of Google Chrome (spanning five years of release history). Then, we studied the technical debt involved with feature toggles by mining a spreadsheet used by Google developers for feature toggle maintenance. Finally, we performed thematic analysis of videos and blog posts of release engineers at major software companies in order to further understand the strengths and drawbacks of feature toggles in practice. We also validated our findings with four Google developers. We find that toggles can reconcile rapid releases with long-term feature development and allow flexible control over which features to deploy. However they also introduce technical debt and additional maintenance for developers.

1. INTRODUCTION

In recent years, one of the top priorities of many companies and organizations has been to re-engineer their release process in order to achieve continuous delivery of new features or at least more timely releases [4, 27]. While traditional releases would take months, modern software companies have managed to reduce their apps' "cycle time" to 6 weeks (Google Chrome [39] and Mozilla Firefox [40]), 2 weeks (Facebook Mobile app [36]) or even daily (Facebook web app [36], Netflix [37] and IMVU [20]). Successful migration towards continuous delivery requires streamlining all release engineering activities [2].

One of the most unpredictable release engineering activities is the integration process [6]. During integration, new features and bug fixes are combined with the latest code from other teams (e.g., by merging git branches [8]) to create a coherent new release. Integration is unpredictable because the combination of many changes late in the develop cycle can lead to instability in the software [30]. Stabilization of such changes can take substantial time, leading to even larger gaps between the latest development and the release stabilization branch. These incompatibilities ("merge conflicts") only surface at integration or testing time, which is close to release time, and hence introduces delays. Such delays are incompatible with continuous delivery.

Although substantial research has been done on predicting painful merges and quantifying the effort involved in a merge [9, 12, 14, 41], release engineers of Facebook, Google and Netflix in their talks at the *2nd International Workshop on Release Engineering (RELENG)* [1] instead declared that they simply try to merge more rapidly, effectively integrating partial (work-in-progress) versions of a feature into the code base instead of waiting for the complete feature. When we asked these workshop participants, "if you are integrating incomplete features, how do you prevent them from interfering with other features?", they unequivocally replied "by using feature toggles"[1].

A feature toggle is an age-old and conceptually simple concept [4, 33]. It basically is a variable used in a conditional statement to guard code blocks, with the aim of either enabling or disabling the feature code in those blocks for testing or release. For example, in Figure 1c, the `if` statement checks the value of the Google Chrome toggle `kDisableFlashFullscreen3d`. If the toggle is present in the Chrome configuration, the `return` statement ensures that the remainder of the method is not executed, effectively disabling the 3D flash feature. In contrast to traditional compile-time feature toggles [7, 15, 16] that exclude features from an application's binary altogether, modern feature toggles allow features to be switched on or off without recompilation, i.e., at run-time or at least at startup-time.

Despite the conceptual simplicity of feature toggles, they also contain risk. Each toggle basically "comments out" large blocks of code (features) that are not yet ready for testing, release or should be used only by a small number of users. This leads to partially dead code and hence technical debt, unless features are made permanent (by removing their `if`

[1]Sometimes also called "flippers", "switches" or "gates".

MSR'16, May 14-15, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2901739.2901745

statements) as soon as they are stable. Furthermore, as the number of feature toggles grows, there is a combinatorial explosion of possible feature sets that need to be tested, as an organization could pick any combination of features for their next release. Large companies such as Facebook, Yahoo, Google and Adobe [18] are cognizant of some of these problems, but similar to small and medium sized software companies, it is unclear to them what is the long-term impact of feature toggles and what best practices exist.

Although toggles are used extensively in industry, we are unaware of any empirical study that examines how feature toggles actually are being used in the software industry. Hence, we conducted a mixed-methods study to provide a better understanding of feature toggles and their benefits and risks.

In the first stage, we measure the following basic parameters in an exploratory study of toggle use on Chrome:

1. Adoption: How many toggles exist?
2. Usage: How are toggles used?
3. Development Stage: Are toggles modified during development or release stabilization?
4. Lifetime: How long do toggles remain in the system?

In the second stage, we examine toggle maintenance and debt by studying the impact of a toggle maintenance campaign launched by Chrome developers. In the third stage, we complement and generalize our Chrome results by examining the talks and writing of prominent release engineers of large successful companies including Twitter, Facebook, Google, etc.

The paper is structured around these research stages. In section 2, we define feature toggles and give examples of actual toggles on the Google Chrome projects. In section 3, we describe our mixed methods research approach and the three data-sources that we use in our study. In section 4, we conduct an exploratory case study to measure the adoption, usage, development stage, and lifetime of toggles. We also discuss the lifecycle of toggles on Chrome. In section 5, we describe toggle debt and the toggle maintenance campaign and the types of toggles used on Chrome. In section 6, we use the talks and blog posts of developers on other large successful projects to generalize our understanding of toggle use. In section 7, we discuss threats to validity. In section 8, we conclude the paper, tying together our findings from our various data sources, and we discuss future work.

2. TOGGLE BACKGROUND AND EXAMPLE

According to Martin Fowler, there are two types of feature toggles: business and release toggles [21]. A business toggle is used to "selectively turn on features in regular use", without requiring re-compilation. The most basic way to do this at program start-up is via command line switches, while more advanced systems allow features to be turned on at run-time. For example, the Google Chrome browser provides the *chrome://flags* web page that allows some of the features to be enabled or disabled without recompilation.

In contrast to business toggles, release toggles are a relatively recent phenomenon that Fowler describes as originating

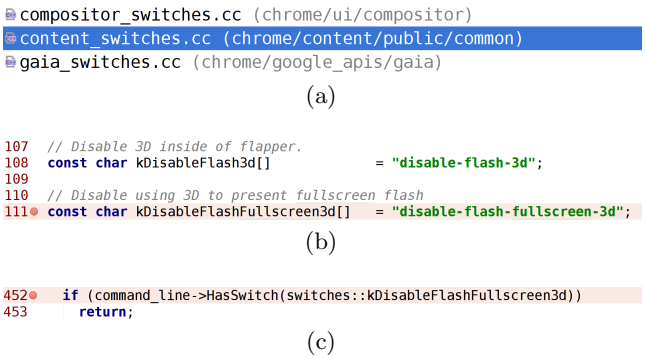

Figure 1: Examples of Chrome 23, showing (a) the names of known switch files, (b) example toggles inside `content_switches.cc`, and (c) code that is covered by the toggle `kDisableFlashFullscreen3d`.

from the move towards continuous delivery of software systems [27]. With increasingly shorter release cycles, features are developed behind release toggles such that they can easily be disabled if the feature is not yet ready for release. This flexibility allows release engineers to disable a feature that is blocking a release simply by disabling the feature toggle. In addition, Bass et al. [4] promote the use of release toggles to decouple the roll-out of a new feature to data centers from the actual release of the feature to (subsets of) users. In this paper, we focus on both kinds of feature toggles.

From a practical perspective, Fowler states that feature toggles require the following [21]:

1. A configuration file with all feature toggles and their state (value).
2. A mechanism to switch the state of the toggles (for example, at program startup or while the application is running), effectively enabling or disabling a feature.
3. Conditional blocks that guard all entry points to a feature such that changing a toggle's value can enable or disable the feature's code.

To illustrate Fowler's requirements, we use Google Chrome's toggle implementation. Chrome is a popular, large open source project that has been using feature toggles for more than five years and that we use as case study in this paper. Instead of having a single configuration file for toggles, Chrome has multiple "switch files" representing toggleable sets of related features, see Figure 1a. For example, there are switch files for features related to how content is displayed in Chrome (`content_switches.cc`) or to GPU testing (`test_switches.cc`).

Each switch file contains a set of feature toggles that can be enabled or disabled. In Chrome, feature toggles are strings that can be set or unset. For example, Figure 1b shows the definition of the toggle `kDisableFlashFullscreen3d`. When activated, this toggle disables the feature that renders graphics in 3D when flash content is presented full-screen. Advanced users might use this toggle to improve performance [5]. These feature toggles are then used inside conditional statements, as shown in Figure 1c. If the toggle `kDisableFlashFullscreen3d` has been set a `return` statement exits the method without executing the 3D rendering feature.

3. METHODOLOGY AND DATA

In order to address our research questions, we use a mixed methods research approach that uses three separate data sources and methodologies [42]. Since there is little written in the scientific literature on feature toggles, we first conduct a quantitative case study on Google Chrome to answer basic quantitative research questions such as the prevalence and lifetime of feature toggles. At the time that the study was conducted we were not able to collect other opend source project that are using feature toggles. In discussions with Chrome developers, they told us of a toggle maintenance campaign in which toggles were categorized to determine if they were technical debt. The campaign resulted in a spreadsheet [28] that we were able to mine to get a more fine-grained understanding of the types of toggles.

To analytically generalize our quantitative case study findings [43] and discuss them in a larger context, we used thematic analysis to code the talks and writings of well-known release engineers working at large successful companies. This analysis helped us to expand and triangulate our single case study with descriptions of how practitioners at other companies use toggles, as well as to identify costs and benefits of using toggles. Finally, we also employ member checking to validate both our qualitative and quantitative findings with four Chrome developers. We now describe the methods and data used in our study in more detail.

3.1 Mining Chrome's Version History

First, we conduct an exploratory case study of the Google Chrome project to quantify the use of toggles. We selected the Chrome browser because it extensively uses toggles, in total we found over 2.4 thousand distinct toggles. Since most of the developers on Chrome are paid by Google and work in Montreal, we were able to member check our findings with Google developers who also use toggles internally on Google projects.

We mined the Google Chrome development history from 2010 to 2015, which covers 39 releases from release 5 to 43. Our main data source was the git version control system of Chrome, in which official releases are tagged. To identify toggles, we manually examined the Chrome source code and found that developers manage such toggles in files whose name ends with `switches.cc` and `switches.h` (as Chrome developers refer to toggles as either switches or flags). Each individual toggle is defined in a switch file. We then parsed the source code to identify how often each toggle is used. For each commit, we determined changes in toggles, i.e., addition, removal and modification of toggles. We recorded the date of these changes to allow us to extract the chronological order of toggle changes.

We provide basic quantitative results in terms of how many toggles are in use as well as more sophisticated analyses such as a survival curve indicating how long toggles survive across time.

3.2 Mining Toggle Maintenance Spreadsheet of Chrome

A toggle maintenance campaign was launched on Google Chrome at release 35. This maintenance effort gave us a unique opportunity to study the technical debt associated with toggles (from the perspective of Chrome developers) and to get a more fine-grained classification of toggles. It also helped to explain anomalies that we observed in our version history data at release 35.

We based our analysis on the official toggle maintenance spreadsheet created by Chrome developers in March 2014, and still maintained at the time of writing [28]. For each toggle, this spreadsheet contains its name, the feature set file name, the owner of the toggle, the status of the toggle ("To keep", "To Remove", "Removed", "Keep", and "Untriaged"), any associated bugs, and a comment from the owner about the purpose of the toggle. By examining the latter "purpose of toggle" field, we were able to identify three major toggle types: long-term, development and release. We created a survival curve for each toggle type, showing how many toggles had a lifetime of 0, 1, 2, etc. releases.

3.3 Thematic Analysis of Practitioners' Talks and Writings

Feature toggles come from industry, not research, hence there is very little research literature on toggles. Practitioners tend to share ideas via conference talks and short blog posts instead of writing detailed technical research papers. We collected data from the recorded talks of sixteen prominent release engineers at 13 large successful companies. To select this material, we examined all the talks and short papers from the three editions of the RELENG international workshop on Release Engineering [1]. We then performed individual online searches with the following four terms: "feature toggles", "feature flags", "feature switches", and "feature flippers."

After eliminating irrelevant content (i.e., non-SE hits and pages only mentioning feature toggles in passing), our final list included 17 talks and blog posts from the following 13 companies: Google, Facebook, Yahoo, Flickr, Netflix, Lyris, BIDS Trading Technologies, Indeed.com, Harel-Hertz Investment House Ltd., IMVU, ThoughtWorks, Rally Software and Adobe. In the references for each of these talks and blog posts, we mention the job title of the author to indicate his or her level of expertise. For example, we examined talks from Paul Hammond [3], who runs the Engineering group at Flickr, and Chuck Rossi [36], who leads the release engineering team at Facebook.

To uncover emergent abstract themes in the blog post and talks, we used a simple thematic analysis to extract the main uses, benefits, and costs noted by these engineers [22, 42]. The specific steps we used are as follows:

1. First, we printed blog posts and paraphrased talks. For the talks, we included timing information to have a direct link to the original evidence.

2. To code the artifacts, we wrote notes in the margins. We cut the printed material to divide it into separate groups of emerging codes. We then progressively repeated this grouping and comparison step, attaining more abstract codes that we recorded on memo pads.

3. We continued to sort, compare, group, and abstract codes until we obtained a set of high-level themes that can each be traced back to practitioner statements, and explain how practitioners perceive the use of feature toggles.

3.4 Member Checking of Findings

Finally, we validated both our qualitative and quantitative results using member checking [42]. We sent the manuscript to four Google Chrome developers to check that the number

of toggles we were reporting agreed with their intuition. We also met with them in person. One of the developers pointed out an error in our understanding of a toggle's state, which lead to a correction in our measures.

4. EXPLORATORY QUANTITATIVE STUDY OF TOGGLES ON CHROME

In this section, we conduct an exploratory case study of Google Chrome to quantify the basic characteristics of toggles. To give context to our quantitative results, we also describe the life cycle of a toggle according to our discussions with Chrome developers. We quantify the following:

1. Adoption: How many toggles exist?
2. Usage: How are toggles used?
3. Development Stage: Are toggles modified during development or release stabilization?
4. Lifetime: How long do toggles remain in the system?

4.1 The Lifecycle of a Toggle on Chrome

Feature toggles were introduced to Chrome development to help with release slips. The goal was to integrate all code up front, then disable code on the release branch that was not yet ready for release [30]. Toggles have the following life cycle:

1. With each new feature a toggle is created to allow developers to enable or disable the feature.
2. By default, the code for the new feature is merged into the organization's common development branch (trunk), but the feature is toggled "off". Toggles allow the new feature code to be merged with the existing code, but without necessarily having the feature active (being tested).
3. When the developer feels that the feature is working, the feature is toggled "on" by default on the development trunk, making the feature active in all compilations and tests.
4. Every 6 weeks, a new release stabilization branch is created from the development trunk. The release team discusses each new feature with the author to determine if the feature is ready for release. Only those features that are ready for release stabilization are toggled "on" for the stabilization branch (and which will likely be released to the public).
5. If a feature becomes unstable during stabilization testing and use, the feature is toggled "off" on the stabilization branch. In such cases, toggles allow the feature code to remain deactivated in the code base.
6. After a feature is released and has proven to be stable in production, either (1) the toggle itself and any old feature code that is no longer necessary can be removed, or (2) if the old and new feature code satisfy different business use cases, the toggle code is kept. Case (1) happens in case of a release toggle, while case (2) happens for a business toggle.

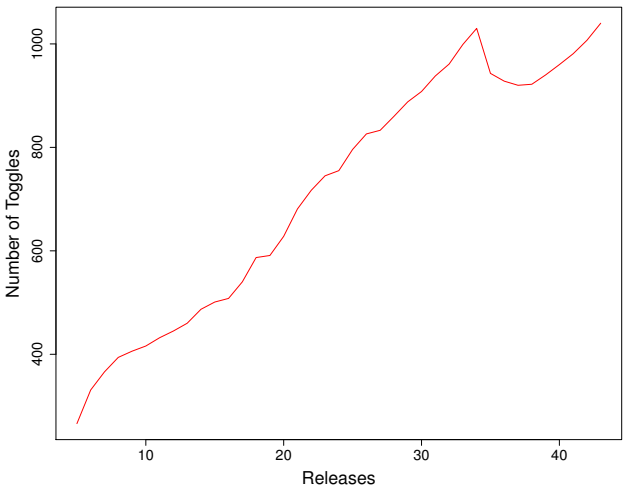

Figure 2: Number of unique toggles per release of Google Chrome.

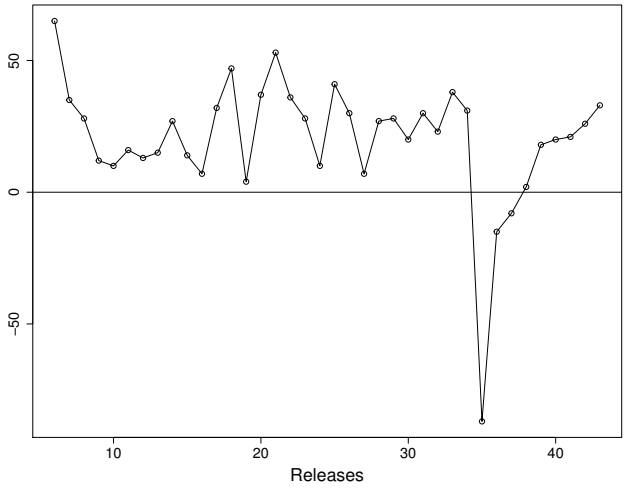

Figure 3: Change in number of unique toggles across releases of Google Chrome. A negative value means that a release has less toggles than the previous one.

Given that feature toggle may be removed after it is deemed that the feature is ready, not all features will therefore have a respective toggle in a release. If two features are interdependent, the corresponding toggle should preferably be interdependent as well.

4.2 Adoption: How many toggles exist?

Figure 2 shows that the number of toggles has grown rapidly since their introduction. At Chrome's 5th release there were 263 toggles compared to 1,040 toggles at the 42nd release. Across all 39 releases, a total of 2,409 distinct toggles have been used, 70% of which were removed at some point. To understand the evolution of the number of toggles, Figure 3 shows for each release the relative change in number of toggles compared to the previous release.

The total number of toggles is growing with a median increase of 30 toggles per release. Each release sees a median of 73 added and 43 removed toggles. The increase initially

fluctuates around 15 toggles per release, then varies between 3 and 50, before stabilizing on about 25 new toggles per release right before release 35. It is clear that the maintenance effort started in release 35 had an immediate effect reducing the number of overall toggles. However, the campaign to remove obsolete toggles lost its momentum at release 39 where the number of toggles returned to historical levels.

From these numbers, it is clear that after their introduction by Chrome release Engineer Laforge [30], toggle use grew rapidly as developers were required to place new features behind toggles. The increase of toggles and the long term ineffectiveness of the toggle maintenance campaign indicates a potential for toggle debt in the form of obsolete toggles, which we discuss later.

The number of toggles increases linearly over time, except for a period of active maintenance at release 35.

4.3 Usage: How are toggles used?

Toggles can be used directly in a conditional statement to guard a code region related to a particular feature, or can be assigned to a variable that is used later on in some conditional statement. The direct usage in a conditional, such as an if statement that enables or disables a section of code, is the most common (see Figure 4). Such toggles account for between 66 and 70 percent of the total toggles contained in a release.

Toggles can also be assigned to a variable to allow for more flexible and complex usages. For example, lines 454 to 456 of Figure 5 show how the state of two toggles (obtained through a call of the method "HasSwitch") is disjuncted together and assigned to a variable. The resulting variable repeatedly can be used to adjust the behaviour of the system. Alternatively, in Figure 6 the result of a logical toggle expression is returned as is.

These variable assignments and helper methods account for more than 30% of the toggle usages in a release and are more complex than direct use in a conditional. The most complex example that we found involved the assignment of a toggle to a member variable in a class. The class's objects then behave depending on the state of the toggle. Chrome developers confirmed these complex usages and also noted that toggles can be used to change or remove entire Chrome extensions. Complex toggle usages make it difficult to determine the sections of code covered by toggles and advanced dynamic parsing techniques are necessary to fully understand the benefits and risks of toggle that are assigned to variables.

Most toggles are used directly in a conditional, however, at least 30% of toggles are assigned to a variable to allow them to significantly change the behaviour of Chrome, which complicates maintenance.

4.4 Development Stage: Are toggles modified during development or release stabilization?

Google releases new versions of Chrome every six weeks [39]. At the end of each development cycle, a feature freeze is imposed (no more new feature development) and a small

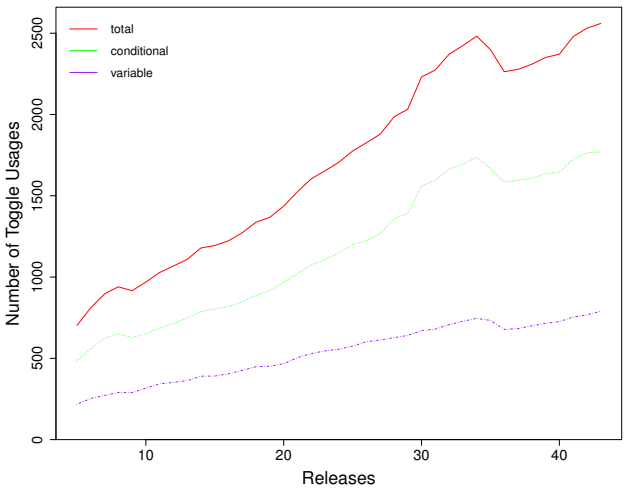

Figure 4: The total number of times toggles are used per release of Google Chrome as well as the type of usage: direct conditional or variable assignment.

```
453   base::CommandLine* cmdline = base::CommandLine::ForCurrentProcess();
454   config.disable_auto_update =
455       cmdline->HasSwitch(switches::kSbDisableAutoUpdate) ||
456       cmdline->HasSwitch(switches::kDisableBackgroundNetworking);
457   config.url_prefix = kSbDefaultURLPrefix;
458   config.backup_connect_error_url_prefix = kSbBackupConnectErrorURLPrefix;
459   config.backup_http_error_url_prefix = kSbBackupHttpErrorURLPrefix;
460   config.backup_network_error_url_prefix = kSbBackupNetworkErrorURLPrefix;
```

Figure 5: Toggles used in a logical expression with the result assigned to a configuration object. The resulting configuration object then is returned in chrome/browser/safe_browsing/safe_browsing_service.cc of release 43.

group of maintainers determines which features are ready to be officially released. Features not ready for release are turned off before the code is sent to the stabilization channel to prepare for production. To determine which toggling events happen during active development and which ones during pre-release stabilization, we separated development commits (before feature freeze) from stabilization commits (after feature freeze) and analyzed which commits change the values of toggles [34].

From the 5k commits that introduce, remove, or change the value of a feature toggle, we found that 97% of the toggling events occur during development, while only 3% occur during release stabilization. If we exclude introduction, removal and renaming of toggles, the percentage of commits with toggling events during development becomes 99%.

These results confirm that toggling is mostly used during development to isolate works-in-progress from other developers, while only some toggling occurs to remove features that became unstable. Since the author and release team meet to determine whether a feature should be enabled for the next release stabilization branch, most features going into stabilization will be stable by design. Since toggles are designed to be disabled, it is much easier to disable an unstable feature than it is to physically revert the set of changes that compose a feature. In the past (in the absence of feature toggles), such reverting had led to a large number of patches during stabilization and delayed releases [30].

```
 99    return command_line.HasSwitch(switches::kInProcessPlugins) ||
100           command_line.HasSwitch(switches::kSingleProcess);
```

Figure 6: Toggle return in chrome/content/renderer/render_process_impl.cc of release 29.

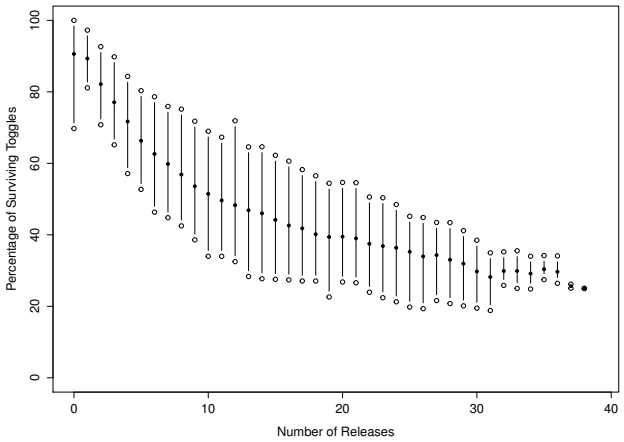

Figure 7: Survival curve showing the percentage of toggles that remain in the code base for 1, 2, ..., 38 releases of Google Chrome. Each vertical bar spans between the 5th and 95th percentile, with the average as the middle point.

Most toggle changes occur during development, only 3% of all toggle changes happen during release stabilization.

4.5 Lifetime: How Long do Toggles Remain in the System?

The Google Chrome developers we talked to measured toggle lifetime in terms of number of releases[2], not days or weeks. As a result, we measure the number of releases a toggle survives in Chrome as the *toggle lifetime*. Figure 7 shows the minimum number of releases a toggle survives for from 1, 2, up to 38 releases, with the X axis representing the number of releases and the Y axis the percentage of toggles that survived at least that many releases (i.e., a toggle that survived for 3 releases is also counted for 1 and 2 releases). Each vertical bar represents the distribution from the 5th percentile to the 95th percentile, while the middle point represents the average percentage of toggles that have survived.

On average 72% of the toggles survive 5 or more releases. At two releases, the survival rate is on average 89%. It takes 6 releases before the average percentage of surviving toggles drops below 70%, 12 releases to drop below 50% and 31 releases to drop below 30%. There is a long tail of toggles that have survived the entire 5 years covered by this study. At 30% the curve stabilizes due to the low number of data points (only release 5 could have toggles surviving for 38 releases, which is the sole data point for X=38). Hence, the averages do not follow a monotonically decreasing trend, especially for the last seven observations (which have 7 or less data points each).

[2]There is a new release approximately every six weeks.

Table 1: Prevalence of each toggle type according to the toggles mentioned in the Chrome spreadsheet [28].

Type	Count	Goal
Development toggles	254	testing and debugging
Long-term business toggles	253	advanced functionality and platform variability
Release toggles:		work-in-progress toggles
- in use	160	guarding unstable features
- cleaned up	51	features have become stable
- technical debt	44	debt, not yet removed

On average 18% of toggles added for a release were removed before the release shipped, while for some releases up to 69% of toggles did not survive the release. The Google Chrome developers confirmed our findings, however they indicated that not all toggles are equal. As we will discuss later, some toggles are intended for long-term use, *e.g.,* they represent some toggleable business function, others are intended to be removed after the feature has stabilized, and others are introduced for a period of less than a release to isolate works-in-progress.

In general, the lifetime for toggles is long, with half of the toggles surviving 12 or more releases.

5. TOGGLE DEBT

Having quantified the prevalence and lifetime of toggles, this section uses data about the maintenance campaign that started at release 35 as an opportunity to study the technical debt created by feature toggles and the extent to which this debt was resolved. Furthermore, the results help us understand how Chrome developers discuss and categorize toggles. We based our analysis on the official spreadsheet [28] created by Chrome developers in March 2014 (and still maintained at the time of writing this paper).

From the spreadsheet, we identified three major contexts in which toggles are used in Chrome: (1) development toggles, which included toggles for testing and debugging; (2) long-term business toggles, which toggle different features to different users; and (3) release toggles, which allow for the gradual roll-out of new features to ensure a feature is ready for production.

Development and long-term toggles are the most common kind of toggle. Development toggles are toggles that help developers to easily enable/disable certain features for testing and debugging, as well as generate diagnostics for error handling. In contrast, long-term toggles are used for configuring the platform on which Chrome will be running, business toggles related to privacy settings, and release toggles that have been converted into business toggles.

160 release toggles were marked as currently active, while 51 had been removed during the maintenance campaign and 44 still awaited removal. In contrast to the development and long-term toggles, release toggles guard features that are work-in-progress, are experimental or just workarounds for existing bugs. Although 84 of such toggles had been marked as "To remove" in the spreadsheet (their feature had become permanent), and 11 toggles had been marked as "Removed", only 51 (20% of all release toggles) had actually

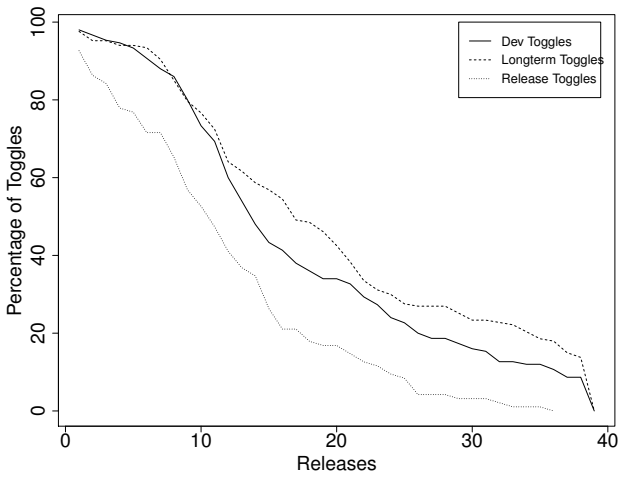

Figure 8: Survival curves for the three different toggle types, showing the percentage of toggles surviving 1, 2, etc. releases of Google Chrome.

been removed, while 44 (17%) still lingered in the source code as technical debt. Interestingly, 2 of the 11 toggles marked as "Removed" were not actually removed yet.

Since, in theory, release toggles should be removed after the feature has been stabilized, their lingering existence is worrisome technical debt. Based on our findings and our discussions with Google Chrome developers, a weakness of release toggles is that it is hard to convince a developer to go back and remove them. There is a lack of tool support for identifying all if-conditions involving a toggle, removing these conditions, integrating the feature code permanently into the surrounding code and ensuring that the (now permanent) feature is still working correctly. On top of the unrewarding nature of the task and lack of tool support, features often take a few releases before they are considered stable and the developer who wrote the toggle code in the meantime typically has moved on to newer features.

Figure 8 shows the survival curves for the toggles of the three types identified in Table 1. 73% of the development toggles and 77% of the long-term business toggles survive at least 10 releases, while for release toggles this is 53%. The curves show that release toggles disappear faster from the code base than the other two kinds of toggles (which intuitively makes sense), however the gap between release toggles and the other toggles is only about 4 releases.

Development toggles roughly have the same trend as long-term toggles until 12 releases, after which they are more aggressively removed. Given their long survival, development and long-term toggles explicitly become a part of the permanent source code (without removing their if-conditions), and hence need to be maintained when the regular code is. Hence, an important part of feature toggles is not being used for not-yet-permanent features, but rather for other reasons.

There are three types of toggles: development, long-term business, and release toggles. Although release toggles are shorter lived than the other types of toggles, 53% still exist after 10 releases indicating that many linger as technical debt.

6. PRACTITIONERS' PERSPECTIVES ON FEATURE TOGGLES

Since toggles originated from industry, there has been little scientific research on them. In this section, we conduct a thematic analysis on 17 talks and blog posts from well-know practitioners. In each reference, we also include the job title of each practitioner to give a sense of his or her expertise [1, 3, 10, 11, 13, 19, 20, 21, 23, 24, 25, 26, 30, 31, 36, 37, 38]. We contrast the themes to our quantitative case study results to analytically generalize our findings and suggest future work. Each section heading represents a theme that emerged.

Reconciling Rapid Release and Longterm Feature Development

The companies that we examined release very frequently. For example, Flickr can release more than ten times a day [3], Facebook releases its website exactly twice a day [36] and Google releases a new major version of Chrome every 6 weeks [30]. To meet these deadlines, these companies limit the scope of features and require teams to continuously integrate their changes into the product's master branch [27]. Small bug fixes are easy to deal with in such a context, but features that take months (and hence span multiple releases) clash with a rapid release strategy. The traditional approach to deal with such features is to have dedicated branches for long-term features, enabling teams to work and experiment in isolation. However, the final merge of the completed feature branch into the master branch can take an unpredictable amount of time, especially when branches have not been synchronized with ongoing development. This merge pain can lead to substantial delay and surprise bugs [29].

For example, Laforge [30], the *release engineer* who designed the six week release cycle for Google Chrome, explains how before the introduction of feature toggles, release branches would be blocked until features were finished. In the meantime, development would build up on trunk, leading to the merging of approximately 500 patches during the release process. Merging such a large number of patches into the release branch in a short time frame introduces instability, causing release deadlines to be missed and developers spending 1 to 3 months stabilizing features for a release (i.e., ironing out show-stopper bugs) instead of focusing on regression bugs.

Therefore, many major companies doing rapid releases prefer to work from a single master branch (trunk) in their version control system and use feature toggles instead of feature branches to isolate feature development (*e.g.,* Google [26] and Facebook [36]). This allows teams to continuously, piecewise integrate their ongoing work on a long-term feature into trunk, but hidden behind a feature toggle, which reduces the total merge effort and makes merging more predictable. Toggles also allow the continuous integration infrastructure [17] to either test the current implementation of a feature or ignore it, respectively by enabling or disabling the feature's toggle. Once the long-term feature is stable enough, its toggle will be enabled by default. Eventually, when the feature has been shown to work well in the field, the toggle can be removed from the code altogether.

Developers at Lyris state that "[toggles] make large features tractable in a continuous build environment" [19], while Harmes, a Flickr developer, declares: "Feature flags and flippers [toggles] mean we don't have to do merges, and that

all code (no matter how far it is from being released) is integrated as soon as it is committed" [25]. This allows web applications like Flickr to deliver new features to users more than ten times per day just by enabling the features' corresponding toggle. In the case of Chrome, feature toggles have reduced the time to stabilize new patches to 11 days [35] (for a 6-week release cycle). This reduction allows release engineers to focus on "stability, security and critical regressions" instead of writing patches for unstable new features [30]. In our quantitative study, we indeed saw evidence that toggles remain in the system for a long period of time, with 72% surviving at least 5 releases.

Flexible Feature Roll-out

The easy-to-disable mentality behind feature toggles affords developers, release engineers, and operation managers enormous flexibility. In particular, feature toggles provide the flexibility to gradually roll out features and do user "experiments" (A/B testing) on new features. For example, "Every day, we [Facebook] run hundreds of tests on Facebook, most of which are rolled out to a random sample of people to test their impact" [11]. If a new feature does not work well, it is toggled off. Otherwise, the feature is rolled out to a larger user base (see below). The ability to flexibly enable and disable feature sets to specific groups of users to determine their effectiveness early on, reduces the investment in features that are not profitable.

Furthermore, feature toggles also facilitate a gradual roll-out (so-called "canary deployment") where an increasing percentage of end users see a new feature, *e.g.,* 10%, 20% up to 100% [4]. If unexpected issues arise as the system scales from hundreds to millions of users (something that cannot be simulated in-house before deployment), servers can be toggled back to the previous version of the system without requiring a new image to be loaded on (virtual) machines [19]. This flexibility is especially important when a company has updated thousands of servers and cannot afford the downtime of loading a new image onto each machine.

Since Chrome is a desktop application, its feature roll-out is more gradual. Features that are ready for release are enabled on the alpha and beta channels and are gradually exposed to a larger population. As we saw in our quantitative results, only 3% of toggle events occur after a feature has made it to release stabilization, indicating that most new features are quite stable and few need to be toggled out.

Enabling Fast Context Switches

Kerzazi et al. [29] found that a large percentage of broken builds, i.e., code bases that do not compile or pass tests after merging a branch, are due to developers checking in their changes into the wrong branch. For example, if a developer working on a given feature in a dedicated branch gets a request to fix an urgent bug, she needs to switch to another branch, fix the bug and test it, then switch back to the original branch to continue feature development. Developers often mistake the branch they are in, leading to commits to the wrong branch.

Ho, a developer at Google [26], explains how feature toggles allow developers to prioritize the features and bug fixes they want to work on. If a higher priority bug fix comes up, one can just disable the toggle of the feature one is currently working on, enable the toggle of the feature to fix, then start fixing the bug. Returning back to the original feature is equally easy, without the need to switch branches and potentially lose uncommitted changes. According to Ho, toggling requires less effort than switching branches, hence it reduces the potential of unwanted check-ins and accompanying broken builds. Rally Software's Scott [38] goes as far as stating "Context switching is a much bigger productivity killer than a single line of added conditional complexity [*i.e.* a toggle] in a source file will ever be."

From our quantitative Chrome results, we saw that over 97% of toggle changes occur during development. This fast context switching by changing a toggle value clearly allows developers to test and isolate work-in-progress and is an important aspects of toggles for Chrome developers.

Features are Designed to be Toggleable

Feature toggles require a shift in development mentality, as they require all features to be easy to revert (disable). For example, the on-call developers at Facebook responsible for monitoring how new features behave in production (DevOps) need to be able to disable malfunctioning features within seconds to avoid affecting millions of users [36].

Developing new features behind a feature toggle requires discipline and additional effort during feature design and development [30]. Features must be isolated from each other to avoid toggle dependencies. Done incorrectly, a feature's implementation could cross-cut a large number of components, all of which would be scattered with feature toggles. Sowa notes that developers should not "go into your existing class and sprinkle code [toggles] everywhere" [19]. For this reason, adding toggles to features that already exist in the system is difficult and not recommended by practitioners.

However, the investment in making features independent and toggleable has the positive side-effect of making the system more decoupled from other features [19]. This side effect is comparable to the side effect of writing unit tests – unit tests force the system to be written in a modular manner because each unit must be tested independently. This investment remains after the feature toggles have been physically removed from the system and leads to lower coupling, which should improve long-term maintenance and testing of a system.

Toggle Debt

In a blog post entitled "Feature Toggles are one of the worst kinds of Technical Debt", Bird describes some of the major pitfalls associated with feature toggles [10]. The core problem that he points out is that permanent code is intertwined with unreleased, potentially failing code.

Dead code: The most commonly cited disadvantage of toggles is that they can be left behind in the code base instead of being removed once a feature is stable [10, 19]. In some cases, temporary release toggles can even be turned into permanent business toggles, for example to limit access to a feature to certain groups of users. In general, a feature's toggle should be removed as soon as possible to reduce the complexity of the code. According to Sowa, "We started using feature toggles a lot and by the end of 2009 we ended up with 80 active toggles in production. In 2010 we decided to clean up the toggles that are no longer in use because 80 toggles were too much and making the process complex." After the clean up, 48 toggles were left [19]. Since Chrome has over 1000 toggles in use, it is clear that the number of toggles a project is comfortable with varies dramatically.

A second case of dead code related to toggles is when a feature that should be removed is left in the code as is, with its toggle disabled. Having new, old, and preliminary code live in the same release is a risky proposition. Bird gives an example of such a failure, when the Knight Capital Group (which produces software for trading companies) accidentally reused an old feature toggle and "when the flag [toggle] was turned on, old code and new code started to run on different computers at the same time doing completely different things with wildly inconsistent and, ultimately business-ending results. By the time that the team figured out what was going wrong, the company had lost millions of dollars" [10].

In the Knight example, tests had not been properly conducted to ensure consistency and compatibility. In contrast, when Netflix migrated from SimpleDB to a Cassandra database, they first built a consistency checking infrastructure to have the new code running in the background with the old code. Once "there were minimal data mismatch (<0.01%) found [...], we flipped a flag [toggle] to make Cassandra as the source of truth" [32].

Preliminary code: Although it is possible to have low quality code checked in behind feature toggles, the practitioners we studied ensured that release toggles that demarcate work-in-progress are of the same quality as any other code checked into trunk, such as bug fixes [19, 30]. For example, on Google Chrome, code that is behind a feature toggle goes through the same testing and review process as any other change made to trunk. That said, whereas branches enable experimentation with temporarily broken code, feature toggles are less forgiving, since code guarded by a toggle at least needs to be compilable.

Combinatorial Feature Testing

Testing is a critical aspect of continuous integration, with large companies like Google and Facebook running a massive test suite on every change made to trunk [36]. As feature toggles allow flexible roll-out of features (see above), any combination of features could become the next release. Hence, in theory every change to trunk should be tested across all possible combinations of enabled feature toggles. This of course introduces an explosion of tests to run, leading practitioners to voice the question "how do we test all possible combinations of features?"

Fowler's advice is to only test the combinations that one realistically is going to use in production [21]. Typically, a product's roadmap is able to select a likely subset of features, of which a handful could still be left out when not ready. For example, the Google Chrome project enables all experimental features on the development trunk to test each change, i.e., only one configuration of feature toggles is being tested at this stage. Then, before the stabilization stage begins, experimental features are disabled and further tests are run on the set of features that are likely to be released. As soon as a scheduled feature is cancelled for the upcoming release, the tests are reconfigured to disable that feature's toggles.

The flexibility afforded by toggles benefits testing. Flexibility during feature roll-out implies that it is easier to spread testing across different groups of users. For example, during beta testing some users could test one combination of features, while others are testing another combination. This in turn represents another way to deal with the combinatorial nature of testing with feature toggles. Our results on Chrome also showed that toggles can in fact support testing activities, with 33% of toggles being used for testing and debugging purposes.

7. THREATS TO VALIDITY

There are a number of specific threats to validity for our study. By focusing on one exploratory case study, our quantitative results suffer from a threat to external validity. As a result, we triangulated and extended our findings beyond Chrome by performing a thematic analysis on talks and blogs by experienced practitioners. Future work is necessary to measure how different organizations, with different kinds of software systems (web or mobile apps) use feature toggles.

For our qualitative findings, we used a simple thematic analysis technique [22]. However, any such study has inherent investigator bias. To reduce this bias, we had multiple authors code the blogs and videos as well as review the final grouping of codes. In the event of disagreement and misunderstandings, we consulted the data or asked Chrome and other developers to reach a consensus for a contentious theme.

There are few threats to construct validity as our metrics are simple quantifications of the time of commits and of changes to toggles made in commits.

Since the Chrome spreadsheet maintenance campaign [28] was a manual effort, its data is not complete and covered a short time frame of Chrome's development. For example, while the spreadsheet was created around release 34, the number of toggles in Table 1 adds up to 724, which would correspond to the time frame around release 20 in Figure 4. We suspect that while the spreadsheet was initially complete, it has not been fully kept up-to-date, likely because of the effort involved. Despite these problems, the dataset is interesting as it is a large sample of toggles that are manually annotated by developers with the rationale for each toggle.

8. CONCLUSIONS AND FUTURE WORK

Feature toggles allow large companies, including Google, Flickr and Facebook, to easily enable or disable new features that are works-in-progress, in order to simplify their merging into the version control system. Although feature toggles are extensively used in industry, as far as we know this is the first in-depth empirical study of toggles. We make three major contributions. First, using Chrome as a case study, we quantify the prevalence and lifecycle of toggles. Second, we categorize the types of toggles and measure the degree of technical debt based on a Chrome toggle maintenance campaign. Third, we describe how practitioners from a variety of large successful companies use toggles. In this section, we integrate our findings and discuss future work.

Reconciling Rapid Release and Long-term Feature Development.

"[toggles] make large features tractable in a continuous build environment" [19]. The companies we studied often release multiple times per day, which clashes with the unpredictable amount of time necessary to merge feature branches as well as the difficulty of reverting such merges afterwards. Instead, feature toggles provide more flexibility. For example, if during the release process a feature is not ready and is blocking a release, a toggle can disable it, unblocking the release. Our quantitative evidence supports the practitioners' views of using toggles for long-term development, as we found that half the Chrome toggles are still in the system

after 12 releases (1.5 years). However, future work is needed to compare the total effort involved in integrating feature branches versus using feature toggling.

Flexible Feature Roll-out.

Feature toggles can change the functionality of the system without recompiling the code. Large web companies use this benefit to gradually roll-out and test the effectiveness of new features, for example using A/B testing to assess the value of features in a live environment. Furthermore, major bugs in new features can be quickly reverted on a web server without the need to revert, recompile, and deploy a new binary. Quantitatively, we found a linearly growing set of feature toggles, except for a period of active toggle maintenance. These sets of toggles allow both developers and end-users to change the features that are being executed (as exemplified by our findings for toggle value changes). Since we studied a desktop application, future empirical studies are necessary to understand how this process works for web apps.

Enabling Fast Context Switches.

Toggles allow developers to toggle off a feature they are working on and switch to a more pressing bug fix, without the overhead of having to switch branches. We found initial evidence of this in the massive number of toggling events that occurred in version control commits, i.e., a developer temporarily committed her work-in-progress, disabled by a toggle. Future user studies are necessary to compare toggle-based context switches to branch-based switches.

Designing Features to be Toggleable.

Designing a new feature such that it can be toggled back to the old feature behaviour requires additional development effort. Features must be as decoupled from each other as possible to reduce the number of toggle dependencies. Done correctly, practitioners state that this effort results in a more decoupled system. Our quantitative findings support this cost, as developers made over 5,044 commits that introduced or re-factored toggles, covering a relatively large number of files and lines. Interesting future work involves studying how toggling affects the architecture of a system.

Toggle Debt.

Old feature code that is disabled through toggles represents "one of the worst kinds of technical debt" [10]. We quantify the maintenance burden of feature toggles by measuring how developers reduce the number of toggles, re-factor feature sets to better organize toggles, maintain naming conventions, change the values of toggles to reflect the state of the system, and document and keep track of existing toggles in a team spreadsheet. Understanding the areas covered by toggles using advanced dynamic analysis techniques is an interesting area of future work.

Combinatorial Feature Testing. The larger the number of feature toggles, the more possible combinations of features must be tested. Fowler's advice is to only test the combinations that one realistically is going to use in production [21]. One third of toggles on Chrome relate to testing and development.

To conclude, feature toggles are a widespread industrial practice and we hope that our empirical investigation will spark interest and future work into the costs and benefits of this important software engineering practice.

9. REFERENCES

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

[2] B. Adams and S. McIntosh. Modern release engineering in a nutshell – why researchers should care. In *Leaders of Tomorrow: Future of Software Engineering, Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Osaka, Japan, March 2016.

[3] O. M. Allspaw and P. Hammond. 10+ Deploys Per Day: Dev and Ops Cooperation at Flickr. In *Velocity: Web Performance and Operations Conference*, June 2009. Job title: Operations Manager & Engineering Manager at Flickr.

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

[5] J. Bauman. Issue 10875074: Ensure we don't use swiftshader to present flash fullscreen - code review. https://chromiumcodereview.appspot.com/10875074/.

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

[7] T. Berger, R.-H. Pfeiffer, R. Tartler, S. Dienst, K. Czarnecki, A. Wąsowski, and S. She. Variability mechanisms in software ecosystems. *Information and Software Technology*, 56(11):1520 – 1535, 2014. Special issue on Software Ecosystems.

[8] 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)*, pages 1–10, 2009.

[9] C. Bird and T. Zimmermann. Assessing the value of branches with what-if analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 45:1–45:11, New York, NY, USA, 2012. ACM.

[10] J. Bird. Feature toggles are one of the worst kinds of technical debt. http://bit.ly/1PP9tGF. Job title: CTO at BIDS Trading Technologies.

[11] A. Bosworth. Building and testing at facebook. http://on.fb.me/1cY6k1a, August 2012. Job title: VP Engineering at Facebook.

[12] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Early detection of collaboration conflicts and risks. *IEEE Trans. Softw. Eng.*, 39(10):1358–1375, Oct. 2013.

[13] C. Cantrell. All about chrome flags. http://adobe.ly/1Aqe4IS. Job title: Senior Experience Development Manager at Adobe.

[14] M. Cataldo and J. D. Herbsleb. Factors leading to integration failures in global feature-oriented development: An empirical analysis. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 161–170, New York, NY, USA, 2011. ACM.

[15] M. Documentation. Conditional compilation in visual basic. http://bit.ly/1C5LYz9.

[16] O. Documentation. Writing device drivers. http://bit.ly/1LWsLWa.

[17] P. Duvall, S. M. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing*

Risk (The Addison-Wesley Signature Series). Addison-Wesley Professional, 2007.

[18] E. Elliott. *Programming JavaScript Applications*. O'Reilly Media, June 2014.

[19] R. L. Erik Sowa. Feature bits: Enabling flow within and across teams. In *Lean Software and Systems Conference*, April 2010. Job title: Director Engineering & Front End Architect at Lyris.

[20] 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. Job title: Technical Lead at IMVU.

[21] M. Fowler. Featuretoggle. http://martinfowler.com/bliki/FeatureToggle.html, October 2010.

[22] B. G. Glaser and A. L. Strauss. *The discovery of grounded theory: Strategies for qualitative research*. Transaction Publishers, 2009.

[23] P. Hammant. Introducing branch by abstraction. http://paulhammant.com/blog/branch_by_abstraction.html, April 2007. Job title: Consultant at ThoughtWorks.

[24] E. Harel. Feature flags made easy. http://techblog.outbrain.com/2011/07/feature-flags-made-easy, July 2011. Job title: Managing Director at Harel-Hertz Investment House Ltd.

[25] R. Harmes. Flipping out. http://code.flickr.net/2009/12/02/flipping-out/, December 2009. Job title: Senior Frontend Engineer at Flickr.

[26] I. Ho and L. Pasricha. Nyc tech talk series: Testing engineering @ google & the release process for google's chrome for ios. https://www.youtube.com/watch?v=p9bEc6oC6vw. Job title: Tech Lead / Manager & Software Test Engineer at Google.

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

[28] P. Kasting. Command-line flag removal status, and how you can help. https://groups.google.com/a/chromium.org/d/msg/chromium-dev/8EjjXUoFqMI/dFqO_M4Yb2gJ.

[29] 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 and Evolution (ICSME)*, pages 41–50, 2014.

[30] A. Laforge. Chrome release cycle. http://www.slideshare.net/Jolicloud/chrome-release-cycle, January 2011. Job title: Technical Program Manager (Chrome) at Google.

[31] A. Mordo. Continuous delivery - part 3 - feature toggles. http://bit.ly/1er1grz. Job title: Search Engine Expert at Indeed.com.

[32] P. Padmanabhan and S. Madappa. Netflix queue: Data migration for a high volume web application. http://nflx.it/1Fk493Z. Job title: Architect & Architect at Netflix.

[33] A. Rahman, E. Helms, L. Williams, and C. Parnin. Synthesizing continuous deployment practices used in software development. In *Agile Conference (AGILE), 2015*, pages 1–10, 2015.

[34] M. T. Rahman and P. C. Rigby. Contrasting Development and Release Stabilization Work on the Linux Kernel. In *International Workshop on Release Engineering 2014*, 2014.

[35] M. T. Rahman and P. C. Rigby. Release stabilization on linux and chrome. In *IEEE Software 2015*. IEEE, 2015.

[36] C. Rossi. Moving to mobile: The challenges of moving from web to mobile releases. Keynote at RELENG 2014 https://www.youtube.com/watch?v=Nffzkkdq7GM, April 2014. Job title: Engineering Director-Release Engineering at Facebook.

[37] B. Schmaus. Deploying the netflix api. http://techblog.netflix.com/2013/08/deploying-netflix-api.html, August 2013. Job title: Engineering Director at Netflix.

[38] R. Scott. Feature toggles - branching in code. https://www.rallydev.com/blog/engineering/feature-toggles-branching-code. Job title: Development Manager at Rally Software.

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

[40] S. Shankland. Rapid-release firefox meets corporate backlash. http://cnet.co/ktBsUU, June 2011.

[41] E. Shihab, C. Bird, and T. Zimmermann. The effect of branching strategies on software quality. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, pages 301–310, New York, NY, USA, 2012. ACM.

[42] J. w Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. SAGE Publications, Incorporated, 2009.

[43] R. K. Yin. *Case Study Research: Design and Methods*, volume 5 of *Applied Social Research Methods Series*. Sage Publications Inc., 3 edition, 2003.