Investigating Modern Release Engineering Practices

Md Tajmilur Rahman

A Thesis

in

The Department

of

Computer Science & Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy (Software Engineering) at

Concordia University

Montréal, Québec, Canada

July 2017

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Md Tajmilur Rahman**

Entitled: **Investigating Modern Release Engineering Practices**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Waiz Ahmed

_____ External Examiner
Dr. Shane McIntosh

_____ External to Program
Dr. Abdelwahab Hamou-Lhadj

_____ Examiner
Dr. Nikolas Tsantalis

_____ Examiner
Dr. Juergen Rilling

_____ Thesis Supervisor
Dr. Peter Rigby

Approved by _____
Dr. Sudhir Mudur, Chair of Department

August 28th, 2017
Date of Defence

_____
Dr. Amir Asif, Dean, Faculty of Engineering and Computer Science

Abstract

Investigating Modern Release Engineering Practices

Md Tajmilur Rahman, Ph.D.

Concordia University, 2017

Modern release engineering has moved from longer release cycles and separate development and release teams to a continuous and integrated process. However, release engineering practices include not only integration, build and test execution but also a better management of features. The goal of this research is to investigate the modern release engineering practices which cover four milestones in the field of release engineering, i. understanding rapid release by measuring the time and effort involved in release cycles, ii. feature management based on feature toggles iii. the impact of toggles on the system architecture, and iv. the quality of builds that contain ignored failing and flaky tests. This thesis is organized as a "manuscript" thesis whereby each milestone constitutes an accepted or submitted paper.

First, we investigate the rapid release model for two major open source software projects. We quantify the time and effort which is involved in both the development and stabilization phases of a release cycle where, we found that despite using the rapid release process, both the Chrome Browser and the Linux Kernel have a period where developers rush changes to catch the current release.

Second, we examine feature management based on feature toggles which is a widely used technique in software companies to manage features by turning them on/off during development as well as release periods. Developers typically isolate unrelated/unreleased changes on branches. However, large companies, such as Google and Facebook do their development on single branch. They isolate unfinished features using feature toggles that allow them to disable unstable code.

Third, feature toggles provide not only a better management of features but also keep modules isolated and feature oriented which makes the architecture underneath the source code readable and

easily extractable. As the project grows, modules keep accepting features and features cross-cut into the modules. We found that the architecture can be easily extracted based on feature toggles and provides a different view compared to the traditional modular representations of software architecture.

Fourth, we investigate the impact of failing tests on the quality of builds where we consider browser-crash as a quality factor. In this study we found that ignoring failing and flaky tests leads to dramatically more crashes than builds with all tests passing.

Acknowledgments

First and foremost, I would like to thank my supervisor Dr. Peter Rigby, for his guidance and support. The past four and a half years working with Peter Rigby has been intense, often challenging, but at the same time very productive and full of learning. I truly admire his dedication to helping his students grow and reach their full potential.

I would like to extend my special thanks to Dr. Emad Shihab, Dr. Wei Shang, Dr. Nicolas Tsantalis who were great mentors to me through the courses I took with them. Also a very special thanks goes to Dr. Bram Adams from Ecole Polytechnique de Montreal for his kind co-operation as a teacher and afterwards as a co-author in one of my research papers.

I would like to thank my dearest colleague, Louis Phillipe Querel for always lifting my spirits up and encouraging me at every steps throughout all these years we worked together. I am also grateful to my other lab mates, Shams Azad, Samuel Donadeli who have made my time at Concordia more enjoyable. Thanks goes to Sumit Sarkar and Alex Moujuri for contributing some intitial work to one of my research papers while we were studying architecture as part of one of our course projects.

Finally I would like to thank my friends and family. Special thanks to Sharmin Sarna for her love and support and staying beside me at each and every stressful moment during every big steps of my PhD. Her motivations helped me a lot to be strong and keep focusing on study. I would like to thank my parents for their well wishes and pure love that always brought blessings upon me. Especially all my gratefulness goes to my mother who was my main inspiration to study in computer science and who always dreamed about my PhD in software engineering.

Contents

List of Figures

List of Tables

Chapter 1

Introduction, Background, and Contributions

In recent years, one of the top priorities of many companies and organizations has been the necessity of re-engineering their release process in order to achieve continuous delivery of new features or at least more timely releases [10, 69]. A study performed by Forester Consulting on-behalf of Thoughtworks [65] shows the majority of the interviewed IT executives want faster release and continuous delivery while the majority of the interviewed business leaders want their software products to bring new features into the market every 3 to 6 months. While traditional releases would take months, modern software companies have managed to reduce their application's "cycle time" to 6 weeks (Google Chrome [128] and Mozilla Firefox [131]), 2 weeks (Facebook Mobile app [114]) or even daily (Facebook web app [114], Netflix [116] and IMVU [54]). Successful migration towards continuous delivery requires streamlining all release engineering activities [3]. My PhD research covers four major aspects of release engineering. Release engineering is a wide area where my studies focus on Time, Quality and Feature Management that cover developer's activity, time and effort spent during each period in a release cycle, feature toggle management, feature toggle architecture, and release quality in terms the number of crashes related to ignoring failing and flaky tests during software build. This thesis is organized as a "manuscript" thesis whereby each milestone constitutes an accepted or submitted paper. Below we provide background

and contributions for each milestone.

1.1 Release Stabilization on Linux and Chrome

One of the most common release engineering activities is integration [13]. During integration, new features and bug fixes are combined with the latest code from other teams (e.g., by merging git branches [15]) to create a coherent new release. Integration is unpredictable because the combination of many changes late in the development cycle can lead to instability in the software [86]. 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.

To compete with the competitors in the market, software companies started adopting shorter release cycles (Rapid Release) instead of longer traditional release cycles. Therefore, new features and activities are published to the end user earlier. This first milestone of my research focusing on time and effort spent in release Linux and Chrome. I also investigate whether developers feel rushed when the release deadline approaches.

Contribution

This empirical study finds that a small teams control the stabilization effort, few changes are reverted, and much of the re-work is not done by the original developer. Despite using regular rapid release cycles, there is a rush period before release stabilization begins. In this milestone, our research includes a deep study on the time and effort involved in a release by comparing two significant segments, development and stabilization, of a release cycle. We have done this study in two parts, first, we study the Linux Kernel alone which is published in Releng2014 [110], and then we replicated the study on Google Chrome and compared the results in these two different types of open source projects practising two different types of development strategies. The extended work has been published in IEEE Software journal in 2015 [111] and is presented in this thesis.

1.2 Feature Management Using Feature Toggles

The second milestone of my doctoral research examines the management of features during the release period. A release manager has a plan of features to publish in the upcoming release versions. When a release deadline approaches, engineers rush their changes into the integration process. Release engineers merge developed features from different feature branches [9] or they just turn on features that are developed on the single branch [26]. In this work, we are the first academic researchers to examine "Feature Toggles" that allows developers to turn off features while development is not finished or turn them on during the testing or during release to opt out/in a feature from public access.

Feature Toggles in Google Chrome

The first part of this study on feature management investigates how feature toggles are used to control the features by software companies. We quantify the use of feature toggles in Google Chrome.

To reduce the probability of integration conflicts major companies including Google, Facebook [114] and Netflix [116] have started using feature toggles to incrementally integrate and test new features instead of integrating the feature only when it is 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, I tried to understand in this part of my study, the prevalence and impact of feature toggles in industry by examining prominent developer blogs and recorded talks.

Contribution

In this part of the research, first, we conduct a quantitative analysis of feature toggle usage across 39 releases of Google Chrome (spanning five years of release history). Then, we study the technical debt involved with feature toggles by mining a spreadsheet used by Google developers for feature toggle maintenance. Finally, we perform a 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.

We did this work with Louis Phillipe Querel, who has a big contribution in this work to calculate the toggle coverage (i.e., how many files are covered under a feature toggle).

This work has been published in the conference of Mining Software Repositories (MSR) in 2016 [109].

1.3 Extracting the Feature Toggle Architecture

We continue our study on feature toggles by investigating the impact on the architecture of the software. Being inspired by the capabilities of toggles to isolate features, we extract the feature view of the architecture based on the feature toggles.

The entire extraction process gives us four different types of architectures: i) the Conceptual Architecture, ii) the Concrete Modular Architecture iii) the Browser Reference Architecture and iv) the Toggle Feature Architecture.

Extract Conceptual Architecture

To extract the modular architecture we follow a traditional approach by Bowman [19]. As part of the extraction process, we extract the conceptual architecture of the system before the concrete modular architecture. We construct the conceptual architectural diagram of Google Chrome from the available documentations.

Extract Concrete Modular Architecture

We extract the concrete architecture of Google Chrome from its source code files and directories. We map the manually derived conceptual modules on to the source code and construct the modular architecture of Chrome browser.

Update Browser Reference Architecture

The extraction of the modular architecture allows us to update the 12 years old browser reference architecture [63] with the current technologies and browser concepts. A reference architecture contains the components required for a particular system domain. New modules are added based on technology and design changes. We also verify that conceptually these modules have been adopted by other browsers for example Firefox and Safari.

Extract Feature Toggle Architectural

Feature toggles not only facilitate developers to manage features while developing on a single branch but also dictate the architecture of the software system. The feature toggle architecture represents the feature dependencies among the modules. Once we identify the feature toggles, we can assign feature dependencies among the concrete and conceptual modules.

We extract the feature toggle architecture of Google Chrome from its source code and show which features touch which modules and which modules contain which features. Feature toggles guard the code for a feature. Using the directory-module mapping derived for the concrete architecture, we extract the toggles contained in each module. Toggles that are contained in multiple modules indicate a feature relationship between these modules.

Contribution

In this part of my thesis we show that toggles can be grouped to identify features which allows us to extract the features perspective of the architecture of a software system. In particular, we extract feature toggle locations in the source code and relate these to the traditional architectural structures.

The outcomes of this work are:

(1) We extract the conceptual, modular and feature architecture of Google Chrome.

(2) Our approach brings out the hidden relationships between modules and features.

(3) We quantitatively analyse how many features a module contains and how many modules a feature spans.

(4) We also investigate how the cross-cutting behaviour of features does evolve over the releases.

(5) We update the browser reference architecture to adjust it with the modern web browsing technologies.

This part of the second milestone of my thesis has audiences from both academics and practitioners. Academics who are interested in teaching or working on web browser technology will have a reference to the major architectural modules as well as a case study of one of the most popular web browsers. Furthermore, developers and researchers who are working on systems with feature toggles will have a method and technique to view the impact of toggles on the modular architecture of a software system.

This work has been submitted to the Journal of Empirical Software Engineering (EmSE). I have performed a the major revision suggested by the editor and am waiting for futher comments. This major revision version is included in the thesis.

1.4 Ignored Failing Tests and Release Quality

We continue our research to investigate the association between software releases and end user crashes. Quality of a release of the software can be measured based on the crash reports. We perform a case study on the popular open source web browser Mozilla Firefox where we investigate the relationship between the reported software crashes and the ignored failing tests during builds. Our main goal of this work is to find the relationship between the browser crash and the build in terms of failing and flaky tests. We investigate the impact of ignoring regular failing tests on the number of crashes. We also investigate whether flaky tests associate the builds with more crashes compared to the builds that do not have any ignored failing test.

Contribution

This research on the association between builds and post-release crashes gives us an understanding about the quality of the builds. The main contribution of this research is, this study opens up the door way to further investigations for example identifying the true causes of the browser crashes or investigating the best approach of minimizing post-release crash by optimizing tests.

We highlight the empirical statistics on the number of builds and crashes on the Development, Beta and Production channels. We find that ignoring test failures increases the number of end user crash for the builds. We also investigate the correlation between the software builds and browser crashes due to ignoring flaky tests. Our findings show that a very high number of crashes are associated to the builds that ignore failing and flaky tests compared to the builds that do not have any ignored failing test.

1.5 Outcomes and Audience

My doctoral research examines the modern release engineering techniques contributing in four areas of release engineering as discussed above. We use successful open source software projects such as Google Chrome, Linux Kernel and Mozilla Firefox for the study. There are four major outcomes of this doctoral research.

- Milestone 1: Release Stabilization, Time and Effort spent in rapid release
    - An investigatory study of release cycles in rapid releases on multiple feature-branches and on single trunk development (Chapter 2).
- Milestone 2: Feature Management with Toggles
    - A case study on modern feature management technique using feature toggles which is the first empirical study ever on feature toggles (Chapter 3).
- Milestone 3: Feature Toggle Architecture
    - A new approach of extracting the software architecture from feature toggles (Chapter 4)

- Milestone 4: Build Quality in Terms of Crashes Associated to Ignored Failing and Flaky Tests
    - An investigation of how the builds associate post release crashes due to ignoring failing and flaky tests. (Chapter 5).

The outcomes of this work is benificial for both researchers and developers as former searches for a deeper understanding of software development as an engineering disciple and the latter searches for best practices that can be applied in their daily work ennvironment.

1.6 Outline

This thesis paper is outlined based on the four milestones and individual research works and publications during the entire study. The following chapter, Chapter 2 describes the study on rapid releases and time and effort spent by developers during different periods in a release cycle. This chapter contains the published work on the Linux kernel and Google Chrome. Chapter 3 contains the study on feature management using feature toggles in Google Chrome which is published in MSR 2016. Chapter 4 contains the research paper on extracting software architecture based on feature toggles and a major revision is currently under review. Chapter 5 contains the investigation on the builds quality in terms of failing tests and end user crashes and will be submitted to the FSE industry track. Chapter 6 discusses the future works that could be derived based on the my research. Chapter 7 concludes the thesis.

Chapter 2

Release Stabilization on Linux and Chrome

Note: this paper was published in IEEE Software [111]. The magazine uses a more familar writing style than is common in academic papers. We include the version before the editor made minor textual changes and reworked our figures to appear in the magazine's format. Those images and minor textual changes are the copyright of IEEE Software.

Abstract: Releasing software involves stabilizing recent development efforts. In this paper we contrast the development period, which involves implementing new features and other major changes, and the stabilization period, which allows for only bug fixes. In the context of the Linux Kernel and Google Chrome projects we characterize development and stabilization in terms of process, developer effort, developer work areas, commit lag time to release, and the number of changes that are rushed into a release. We find that Linux has has a relatively long stabilization period (median 2 months). A small group of 55 and 23 developers control the development and stabilization periods, respectively. For Chrome 10 people take responsibility for 80% of the work during stabilization while 100 people make most of the development changes. On both projects, much of the re-work done during stabilization is not done by the original developer. Both the projects do not allow developers to rush changes into a release.

2.1 Introduction

Large software projects make thousands of changes between releases. During development, new features and other major changes are implemented. Since new changes have been used by relatively few developers and end users, they can have a destabilizing effect on the overall software system. The job of a release engineer is to select and stabilize the changes to a system before it is released to a large user base. This article quantifies the time and effort involved in the release stabilization of two large successful projects – the Linux kernel and the Google Chrome browser. We provide practitioners with our measurement tools,[1] so that they can compare their own projects with Linux and Chrome on the questions listed below. To use the tool, all you need is a Git repository that contains tags indicating the start of release stabilization and the final release. We are willing to help practitioners make these measurements and hope that this kind of grounded empirical findings will help transform software development into an engineering discipline.

How quickly do you release changes?

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

Do you stabilize your own code during a release?

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

Do you rush changes into a release to avoid waiting for the next release?

Nobody wants to wait for the next release, especially if there is only one release per year. As a release date approaches, developers may feel pressured to release features that are not yet stable and well integrated. Chrome switched to a shorter release cycle to avoid pressuring developers into rushing unstable code into a release. Even with these short release cycles, both Chrome and Linux

[1] Measurement tools https://github.com/tajmilur-rahman/measurements

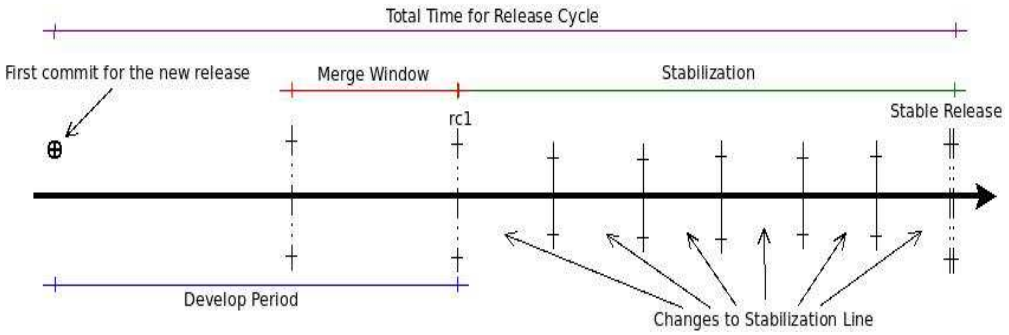


Figure 2.1: Linux release process: Development of subsequent releases occurs in parallel with the stabilization of a release. The two stages join during the merge window where new development is moved onto the stabilization mainline.

see an increase of development work right before release stabilization begins. Some degree of rush seems unavoidable.

2.2 Release Cycle

How rapid are your releases?

> Although Linux and Chrome use regular rapid release schedules, release stabilization varies by approximately 10 days. Over time, both projects have become better at releasing on schedule.

In the early days of Linux development releases were sometimes made more than once per day, prompting Raymond's mantra of "release early, release often" [112]. This trend has continued with many projects adopting increasingly shorter release intervals [78]. For example, Google Plus can release new changes in 36 hours [96] and `Facebook.com` releases twice a day on weekdays [114]. Firefox and Chrome operate on six week release cycles [78, 86].

To quantify the time and effort involved in release stabilization, we use the definitions of the development and stabilization branches as defined by the Chrome and Linux process documents [88, 86]. These branches can be seen in Figures 2.1 and 2.2. The stabilization and release tags in Git allow us to traverse the Git DAG and identify on which branch a change was made. We extract

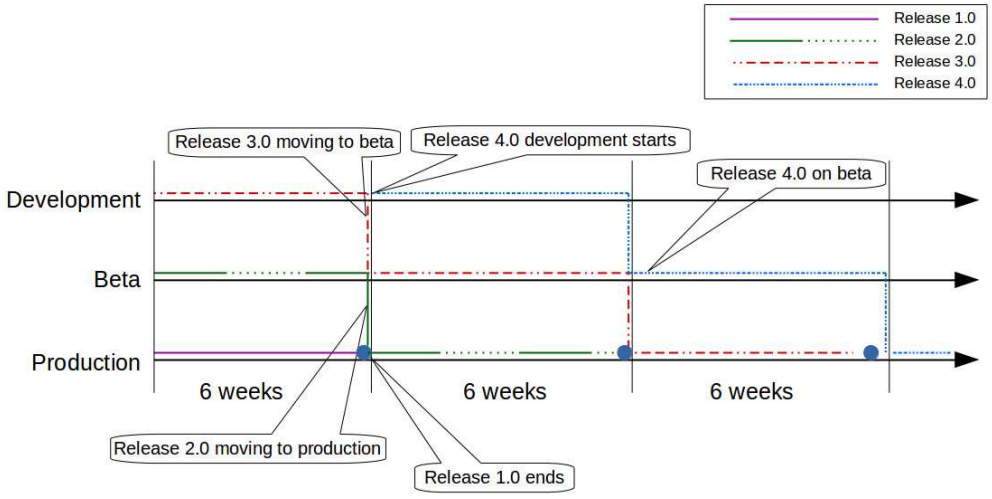


Figure 2.2: Chrome release process: Development occurs in parallel with two stabilization branches – beta and stable. At six week intervals, each branch is moved onto the subsequent stage, for example, development moves to beta.

Churn, i.e., the number of lines added and removed per commit, from the Git version history. We use the Git author field and not the committer field to credit work to developers. For more details on our extraction process see our preliminary work [110].

The **Linux release process** is represented in Figure 2.1. According to the release process documents, Linux uses a flexible time-based release schedule [88], which consists of a merge window and stabilization period. The merge window opens to allow developers to merge changes into the stabilization mainline. The window is open for only two weeks with a standard deviation of 2 days. After the window closes, the first release candidate (rc1) will indicate the start of release stabilization. During stabilization only fixes to regressions and isolated changes, such as device drivers, are merged into the mainline. New release candidates will be created as regressions are found and fixed. We find that on average there will be six release candidates before the final public release. The time period for stabilizing a release continues until no important regressions are outstanding. Stabilization takes on average 62 days (represented by the horizontal line in Figure 2.3) with a standard deviation, minimum, and maximum of 10, 45, and 93 days, respectively. Since release 2.6.31, release stabilization has become more regular. Figure 2.3 shows the variations in the Linux release cycle.

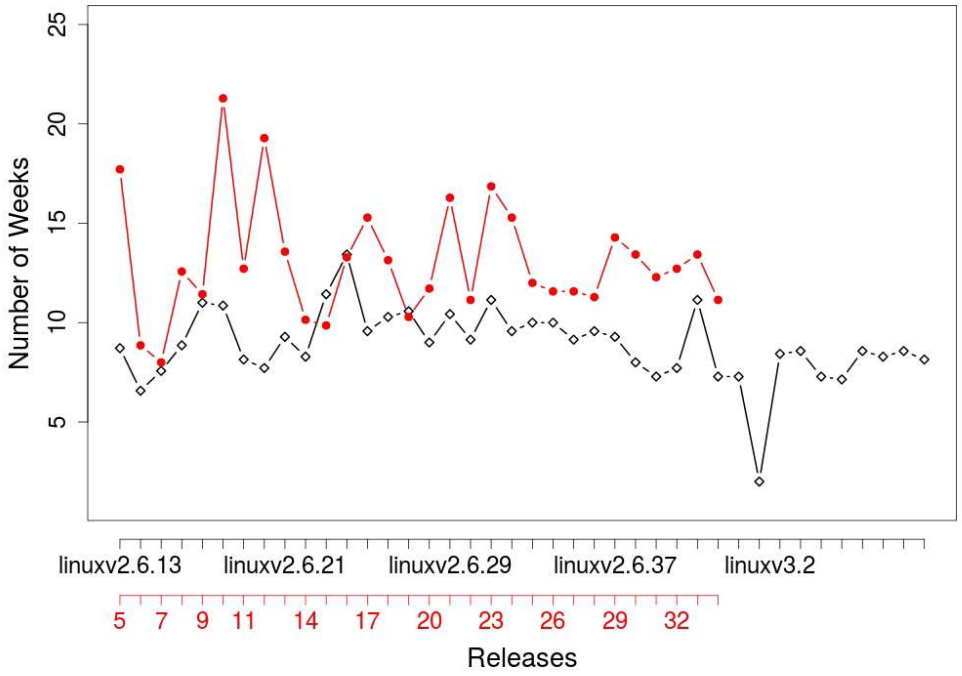


Figure 2.3: *Length of stabilization and development periods for Chrome, upper line, and Linux, lower line. Horizontal lines are expected amount of time for release stabilization.*

Chrome's release process consists of three channels: development, beta, and stable. At six week intervals, the code transitions into the subsequent channel [86]. For example, in Figure 2.2 we see that when development work begins on release 4, release 3 will be moved into the beta channel, release 2 will be moved to the stable channel, and release 1 will be published as a final production release. In our data extraction scripts we are able to identify which channel a commit was made on based on its version number. In this work, we do not differentiate between beta and stable as both channels are related to release stabilization.

We find that release stabilization takes an average of 91 days with a standard deviation, minimum, and maximum of 11, 56, and 149, respectively. Figure 2.3 shows the variations in Chrome's release cycle and the horizontal line shows the ideal 12 week stabilization period. Immediately after adopting a rapid release cycle, there was significant variance in release times with some releases taking substantially longer than 12 weeks. We can see that recent releases have become much more regular.

2.3 Effort

How much effort do you expend in stabilizing a release?

> A very small group of developers control the stabilization of a release – 23 and 10 developers for Linux and Chrome. The majority of changes are made during development with 9% and 7% of lines changed during stabilization, respectively.

We use three basic measures to get a sense of effort involved in developing and releasing Linux and Chrome. We measure the number of commits, churn (number of lines that changed), and the number of people working on the stabilization vs. development branches.

For **Linux** we find that, of the total 381k commits made to kernel source files between 2005 and 2013, 77% of the commits are made during development and 23% are made as part of stabilization. In Figure 2.4, the median development churn per release is 834k lines compared to the stabilization churn of 83k lines. A Wilcoxon test shows that this difference is statistically significant with $p \ll 0.001$. In the median case 91% of the lines changed for a release are made in development with a

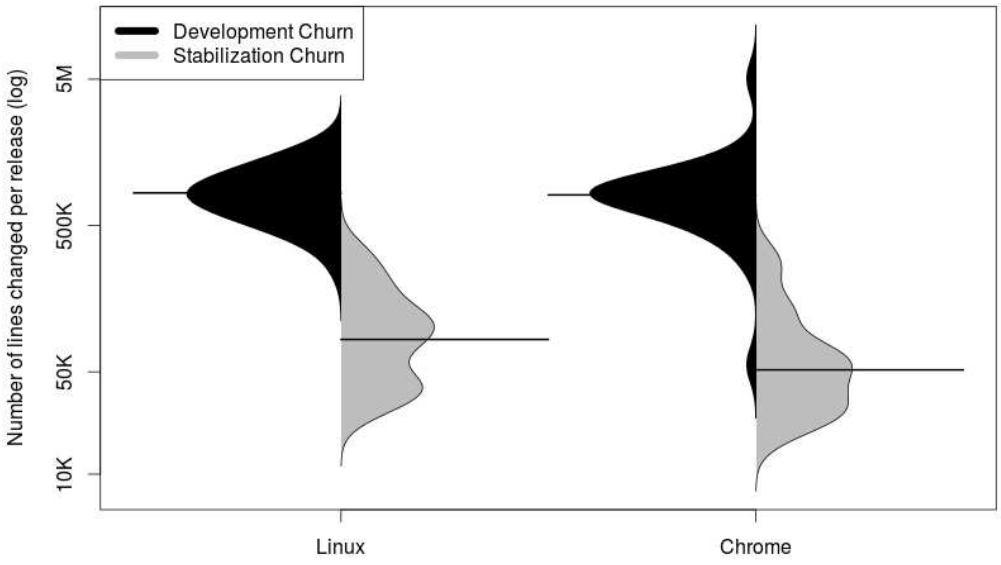


Figure 2.4: *Churned lines of code per release for development and stabilization*

ratio of 105 lines churned per commit, while 9% of lines changed are during stabilization with 41 lines churned per commit. Linux tends to make smaller changes during release stabilization.

For **Chrome** we find that, of the total 164k commits made to source files between 2008 and 2014, 85% of the commits are made during development and 15% are made as part of stabilization. The median development churn per release is 808K lines compared to the stabilization churn of 51K lines (see in Figure 2.4). A Wilcoxon test shows that this difference is statistically significant with $p \ll 0.001$. In the median case 93% of the lines changed for a release are in development with a ratio of 11 lines changed per commit, while 7% of lines are changed during stabilization with 165 lines churned per commit. In contrast to Linux, it is interesting that Chrome release engineers tend to make very large changes during release stabilization.

For the **Chrome team** there are 10 developers that make 80% of the changes during stabilization and 98 developers change 80% of the lines changed during development. For **Linux**, 10K developers have contributed to Linux, however, 55 developers have done 80% of the development work, while 23 developers have done 80% of the stabilization work (See Figure 2.5).

This result is similar to Mockus *et al.*'s [1] finding that the Apache httpd server had a core group of 15 developers who wrote 80% of the code. Linux is a much larger project, we see that during stabilization 23 developers control the stabilization process. Mockus *et al.* noted that as a

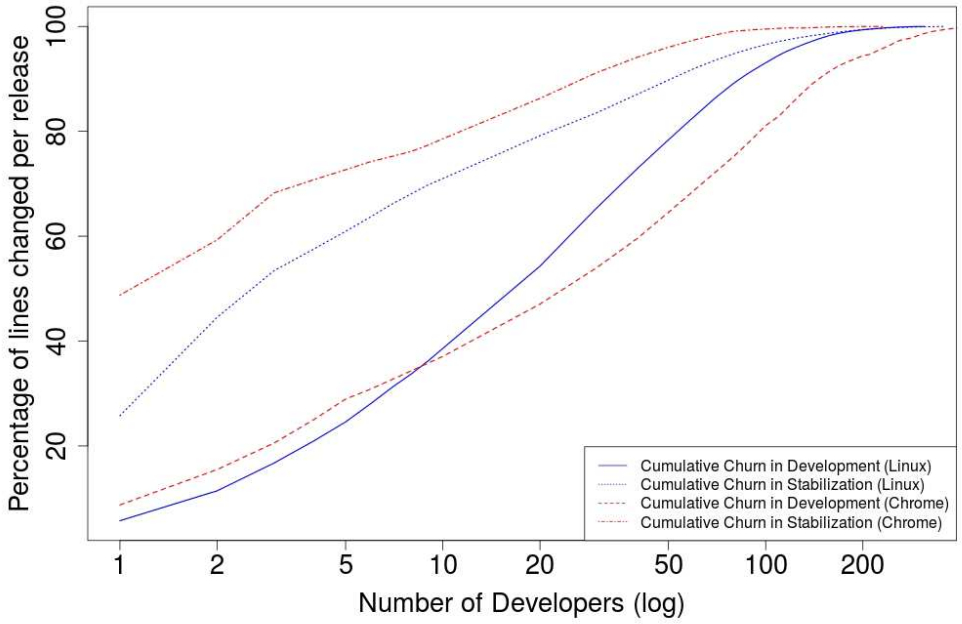


Figure 2.5: *Cumulative distribution of developer contributions*

system grows, *e.g.,* Mozilla, more complex mechanisms must be used to manage it. On the Mozilla project there are 13 release engineers [100]. In order to integrate the development effort from the larger group of 55 developers that account for 80% of the development effort a chain-of-trust is used to pass changes from less trusted developers up to the trusted stabilization mainline that Torvalds controls and makes releases from [88]. Stabilization work occupies the majority of Torvalds's time and clearly represents large contributions from other core developers.

2.4 Ownership

Do you stabilize your own code during a release?

Many developers have their files modified by another developer during stabilization. Few commits are reverted during stabilization.

The Linux Kernel has a policy that 'the original developer should continue to take responsibility

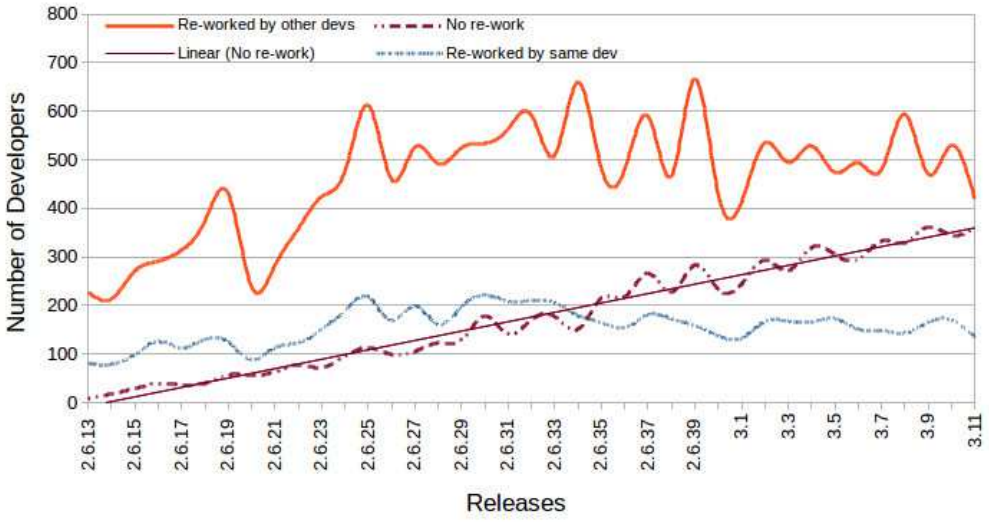


Figure 2.6: *Linux developers re-working files during stabilization*

for the code [they contribute]' [88]. Chrome also has this expectation [86]. We expect developers who modify files during development to fix any problems with those files that arise during stabilization. Of the files that are modified in both periods, we measure the proportion that are done by the original developer vs. those that are modified by other developers and integrators.

For Linux and Chrome, respectively, we find in the median case per release there are 161 and 258 developers who modified the same files they changed during development, 480 and 307 developers who had their files modified by other developers during stabilization, and 171 and 322 developers whose changes did not require any modification during stabilization. These sets of developers are not mutually exclusive. Figures 2.6 and 2.7 depicts this situation for Linux and Chrome, respectively. From these numbers, it would appear that many developers do not take on the responsibility to fix their bugs for a release. Since the number of developers making changes is much larger than the core group of developers, it is likely that many of these changes are made by transient developers who do not remain to fix bugs in their small code contribution. Instead a small group of integrators (See Figure 2.5) is responsible for integration and bug fixes of regressions during stabilization.

An alternative explanation, and threat to validity, is that integrators are working in other areas of the file and are not modifying code lines related to the changes made during development. While a fine-grained, line level analysis is left to future work, it is surprising that the majority of files that

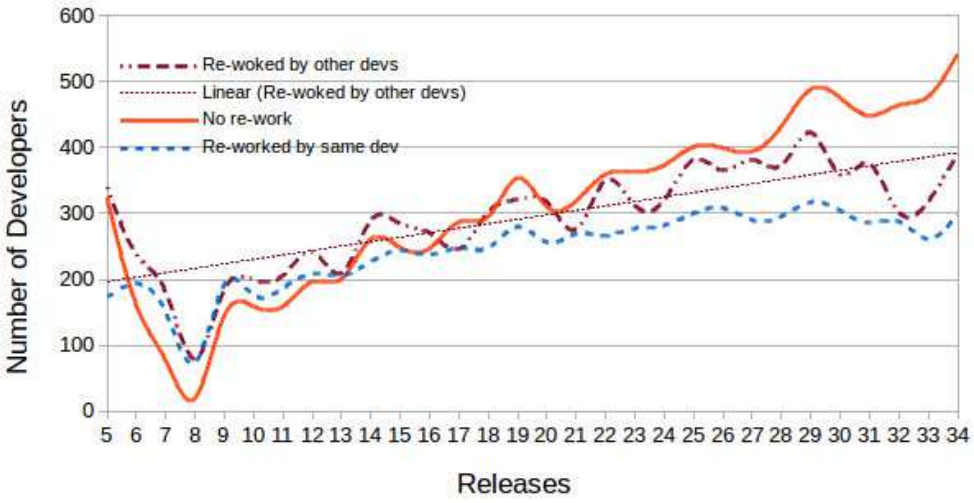


Figure 2.7: *Chrome developers re-working files during stabilization*

need modification during stabilization were modified by a different developer.

For Linux, the amount of re-work done by other developers fluctuates dramatically from 214 to 667 and does not show a clear trend. However, the number of developers whose files do not need to be re-worked shows a clear increasing trend line with an adjusted $R^2 = .97$ and $p \ll 0.001$. This trend likely shows a maturing in the selection of code from external contributors. For Chrome, all three categories are increasing, indicating an increase in the number of developers contributing to Chrome, but obscuring other patterns.

Although a large number of files are modified by release engineers, few changes are reverted during stabilization. For Linux and Chrome, in the median case, there are 104 and 3 reverts per stabilization period. This accounts for only 2.3% and less than 1% of total stabilization commits for Linux and Chrome. For Linux 55% of reverts are made during stabilization, while for Chrome, the majority of reverts, 76%, are made during development.

2.5 Speed

How quickly do you release changes?

Changes made during stabilization (*e.g.,* fixes to regressions) are integrated and released much more quickly than development changes.

We want to understand how quickly bugs are fixed during stabilization and new development incorporated into Linux and Chrome. We define transit time as the number of days it takes for a change to be (1) integrated in the stabilization branch or (2) included in a final release. Previous work measured the transit time for a change to be released [72], but ignored the different purposes of changes and found large variations in transit times (3 to 6 months). By differentiating between change types we find that most of the variation can be explained by whether the change was a fix made during stabilization or a change made during normal development.

In Figure 2.8, for Linux and Chrome, we see that stabilization changes (fixes to regressions) take a median of only 8 days and less than 1 day to be included in a stabilization branch. In contrast, development changes take 35 and 21 days to reach the stabilization branch for Linux and Chrome, respectively. A Wilcoxon test shows that these differences are statistically significant at $p \ll 0.001$.

For Linux and Chrome, the transit time for a stabilization change to be released is 47 and 78 days, while a development change takes 97 to 109 days, respectively. Although Chrome starts release stabilization every six weeks and produces a new release every six weeks, changes to Chrome take longer to reach the user than Linux changes because Chrome stabilizes two releases at a time while Linux stabilizes only one release.

2.6 Rush

Do you rush changes into a release to avoid waiting for the next release?

Two weeks before stabilization begins, the daily churn rate increases by 20% and 21% for Linux and Chrome, respectively.

As a release date approaches, developers may feel pressured to release features that are not yet be stable and well integrated. This pressure increases with long release cycles as developers may

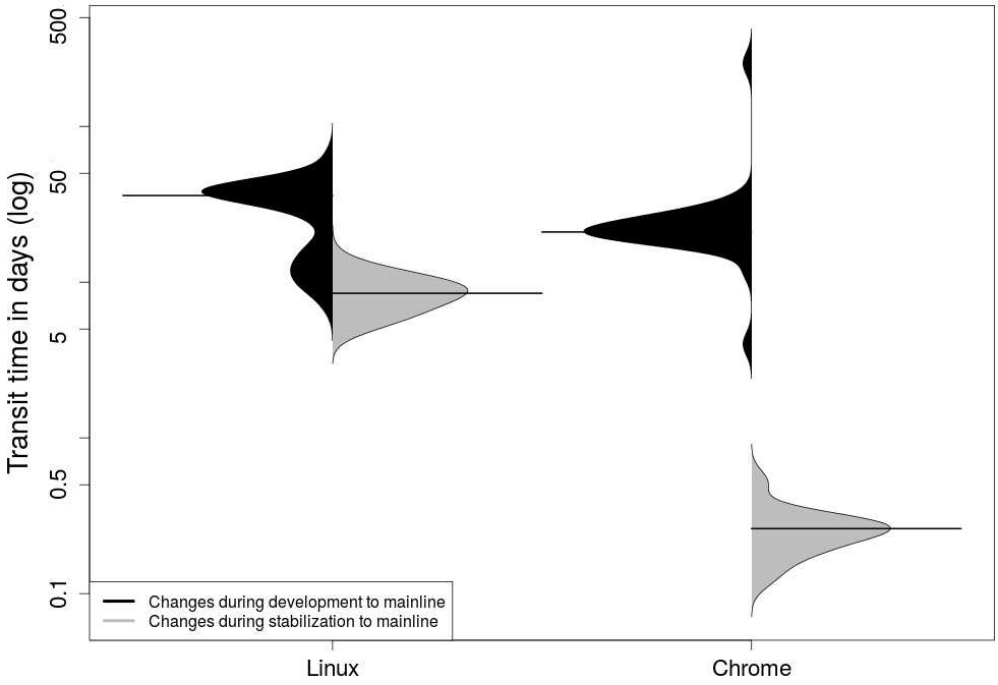


Figure 2.8: *Transit time for the commits to stabilization branch and release*

rush changes into a release to avoid waiting for the next release. Our goal is to empirically test whether developers rush changes in right before release stabilization and feature freeze. To test this, we define the churn rate as the number of lines changed per day. We define the rush period as two weeks before release stabilization begins. The rush period corresponds to the Linux merge window and is one third of the Chrome development period. The normal development period is defined as the period between releases before the rush period begins. On Linux, this is the two months before the merge window opens and on Chrome it is the first four weeks of the development cycle. Since we are interested in development and not integration, we excluded all merge commits. We also used the author date instead of the committer date so that "cherrypicked" changes will be counted during the development period not when they are picked. We hypothesize that the churn rate in the rush period will be higher than the churn rate during normal development.

To test this hypothesis, we use the non-parametric Wilcoxon test to compare the churn rate of the two distributions. For Linux and Chrome, we find a statistically significant difference in daily churn rate between normal development and the two weeks before stabilization begins ($p = 0.007$ and $p = 0.0008$, respectively). In Figure 2.9, we see that in the median case, Linux developers

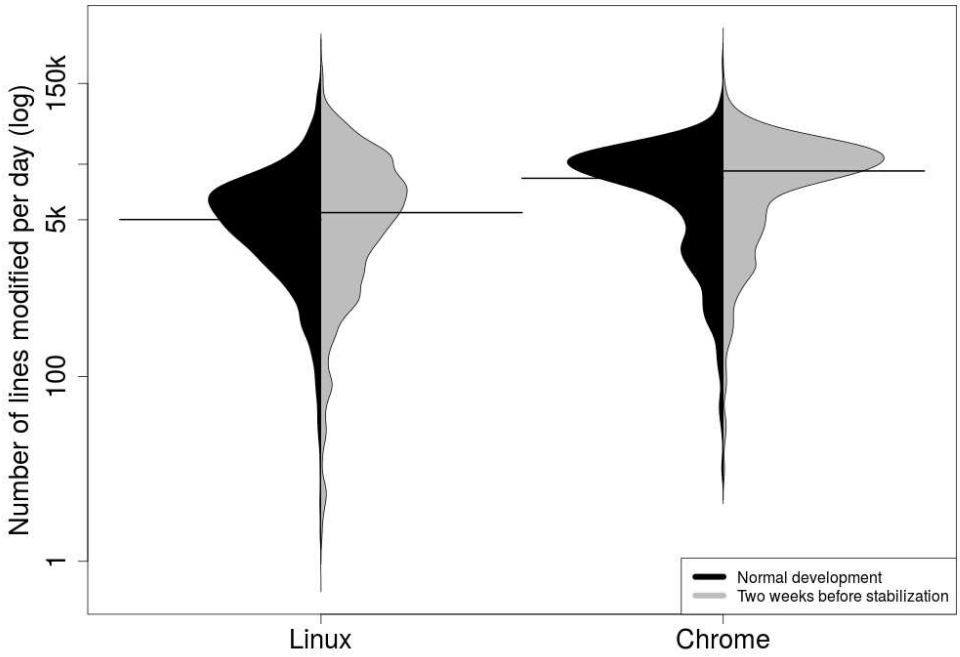


Figure 2.9: *Distribution of daily churn during normal development (left) and two weeks before stabilization (right)*

change 5k and 6k lines per day for normal and rush period respectively. The values for Chrome are 14k and 17k. These differences represent a 20% and 21% increase in median daily churn during the rush period for Linux and Chrome, respectively. Despite having a rapid release cycle, there is still some degree of rush before release stabilization begins.

Crosscutting the time and effort measure we have examined are the three factors that Chuck Rossi, the lead release engineer at Facebook, considers when creating a release: the schedule, the quality, and the feature set [114]. All three cannot be optimized at the same time, so Chuck sacrifices the feature set, but releases stable features on schedule and drops any feature that would reduce quality.

Quality is paramount to Linus Torvalds whose main job is integration and release stabilization. He ranks first in terms of number of integration merges and 52nd in terms of the number of changes made to Linux.

> "I'm not claiming this [change ...] is really any better/worse than the current behaviour from a theoretical standpoint, but at least the current behaviour is _tested_, which makes it better in practice. So if we want to change this, I think we want to change it to something that is _obviously_ better."
>
> –Torvalds, [124]

Likewise, in an article titled "release early, release often," Anthony Laforge, who introduced rapid release to Chrome development, states:

> "While pace is important to us, we are all committed to maintaining high quality releases – if a feature is not ready, it will not ship in a stable release."
>
> –Laforge, [86]

Chrome and Linux value quality over schedule and schedule over features.

Chapter 3

Feature Toggles: Practitioner Practices and a Case Study

Note: this chapter was originally published in MSR'16 as Rahman *et al.* [109].

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.

3.1 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 [10, 69]. While traditional releases would take months, modern software companies have managed to reduce their apps' "cycle time" to 6 weeks (Google Chrome [118] and Mozilla Firefox [119]), 2 weeks (Facebook Mobile app [114]) or even daily (Facebook web app [114], Netflix [116] and IMVU [54]). Successful migration towards continuous delivery requires streamlining all release engineering activities [3].

One of the most unpredictable release engineering activities is the integration process [13]. During integration, new features and bug fixes are combined with the latest code from other teams (e.g., by merging git branches [15]) 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 [86]. 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 [16, 21, 23, 120], release engineers of Facebook, Google and Netflix in their talks at the *2nd International Workshop on Release Engineering (RELENG)* [2] 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 [10, 107]. It basically is a variable used in a conditional statement to guard code blocks, with the aim of either enabling or

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

disabling the feature code in those blocks for testing or release. For example, in Figure 3.1c, 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 [14, 102, 40] 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` 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 [44] 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 subsection 3.1.1, we define feature toggles and give examples of actual toggles on the Google Chrome projects. In section 3.2, we describe our mixed methods research approach and the three data-sources that we use in our study. In section 3.3, 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 3.4, we describe toggle debt and the toggle maintenance campaign and the types of toggles used on Chrome. In section 3.5, we use the talks and blog posts of developers on other large successful projects to generalize our understanding of toggle use. In section 3.6, we discuss threats to validity. In section 3.7, we conclude the chapter, tying together our findings from our various data sources, and we discuss future work.

3.1.1 Toggle Background and Example

According to Martin Fowler, there are two types of feature toggles: business and release toggles [58]. 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 from the move towards continuous delivery of software systems [69]. 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. [10] 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 chapter, we focus on both kinds of feature toggles.

From a practical perspective, Fowler states that feature toggles require the following [58]:

(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.

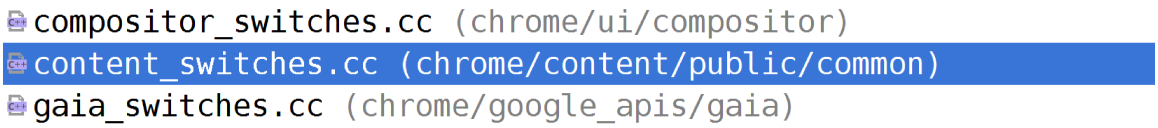
(3) Conditional blocks that guard all entry points to a feature such that changing a toggle's value can enable or disable the feature's code.

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 chapter. Instead of having a single configuration file for toggles, Chrome has multiple "switch files" representing toggleable sets of related features, see Figure 3.1a. 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 3.1b 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 [20]. These feature toggles are then used inside conditional statements, as shown in Figure 3.1c. If the toggle `kDisableFlashFullscreen3d` has been set a `return` statement exits the method without executing the 3D rendering feature.

3.2 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 [128]. 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



(a)

```
107  // Disable 3D inside of flapper.
108  const char kDisableFlash3d[]             = "disable-flash-3d";
109
110  // Disable using 3D to present fullscreen flash
111  const char kDisableFlashFullscreen3d[]   = "disable-flash-fullscreen-3d";
```

(b)

```
529   CommandLine* command_line = CommandLine::ForCurrentProcess();
530   if (command_line->HasSwitch(switches::kDisableFlashFullscreen3d))
531     return;
532   WebKit::WebGraphicsContext3D::Attributes attributes;
533   attributes.depth = false;
534   attributes.stencil = false;
535   attributes.antialias = false;
536   attributes.shareResources = false;
537   context_ = WebGraphicsContext3DCommandBufferImpl::CreateViewContext(
538       RenderThreadImpl::current(),
539       surface_id(),
540       NULL,
541       attributes,
542       true /* bind generates resources */,
543       active_url_,
544       content::CAUSE_FOR_GPU_LAUNCH_RENDERWIDGETFULLSCREENPEPPER_CREATECONTEXT);
545   if (!context_)
546     return;
```

(c)

Figure 3.1: 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`.

campaign in which toggles were categorized to determine if they were technical debt. The campaign resulted in a spreadsheet [74] 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 [131] 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.

3.2.1 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.

3.2.2 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 [74]. 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.

3.2.3 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 [2]. 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 [4], who runs the Engineering group at Flickr, and Chuck Rossi [114], 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 [61, 128]. 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.

3.2.4 Member Checking of Findings

Finally, we validated both our qualitative and quantitative results using member checking [128]. 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.

3.3 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?

3.3.1 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 [86]. 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.

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.

3.3.2 Adoption: How many toggles exist?

Figure 3.2 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 3.3 shows for each release the relative change in number of toggles compared to the previous release.

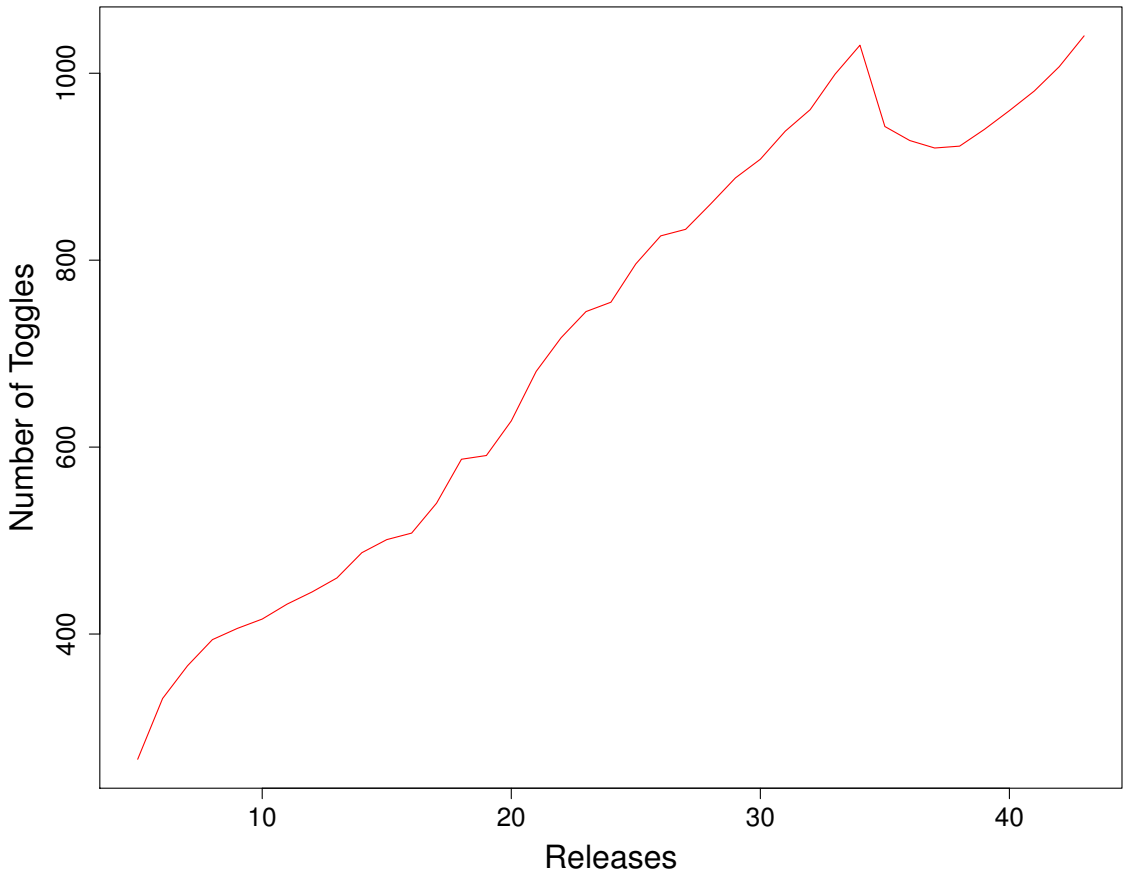


Figure 3.2: Number of unique toggles per release of Google Chrome.

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 [86], 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.

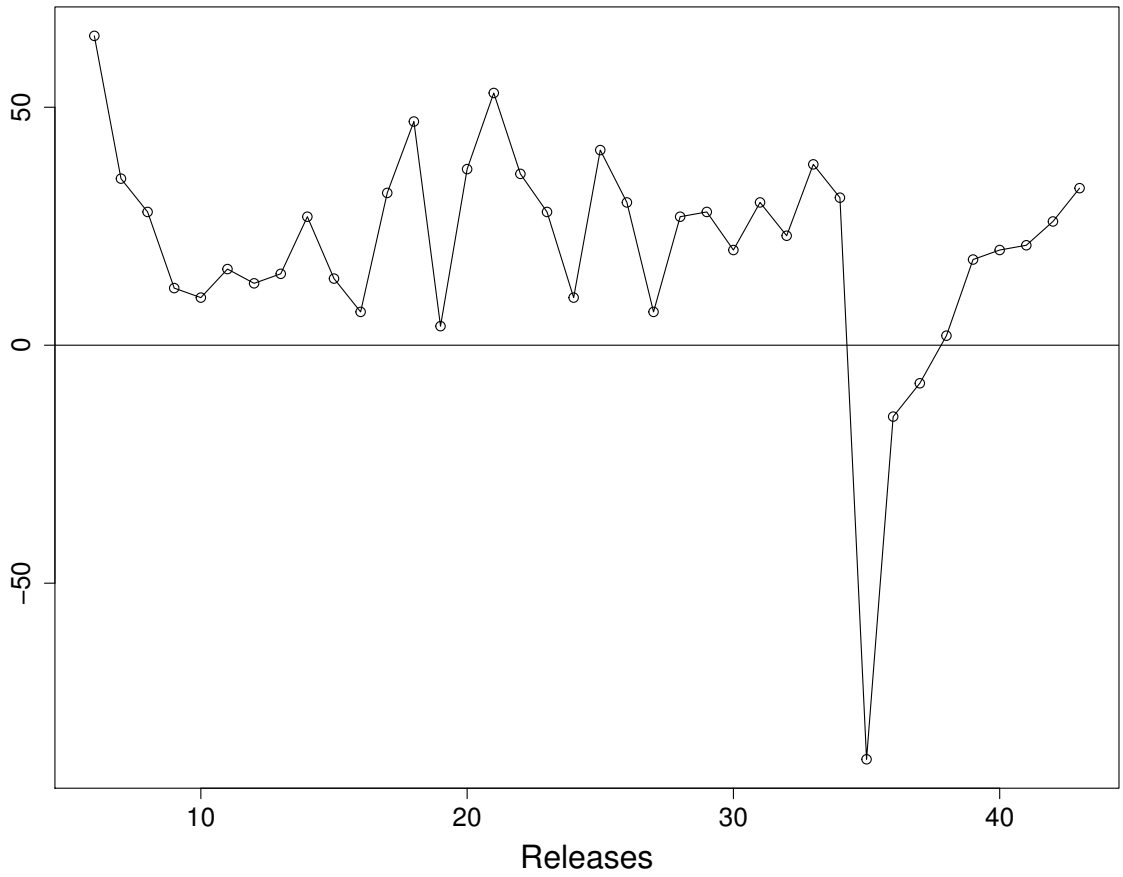


Figure 3.3: Change in number of unique toggles across releases of Google Chrome. A negative value means that a release has less toggles than the previous one.

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

3.3.3 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 3.4). 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 3.5 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 3.6 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.

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

Google releases new versions of Chrome every six weeks [118]. At the end of each development cycle, a feature freeze is imposed (no more new feature development) and a small 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 [110].

From the 5k commits that introduce, remove, or change the value of a feature toggle, we found

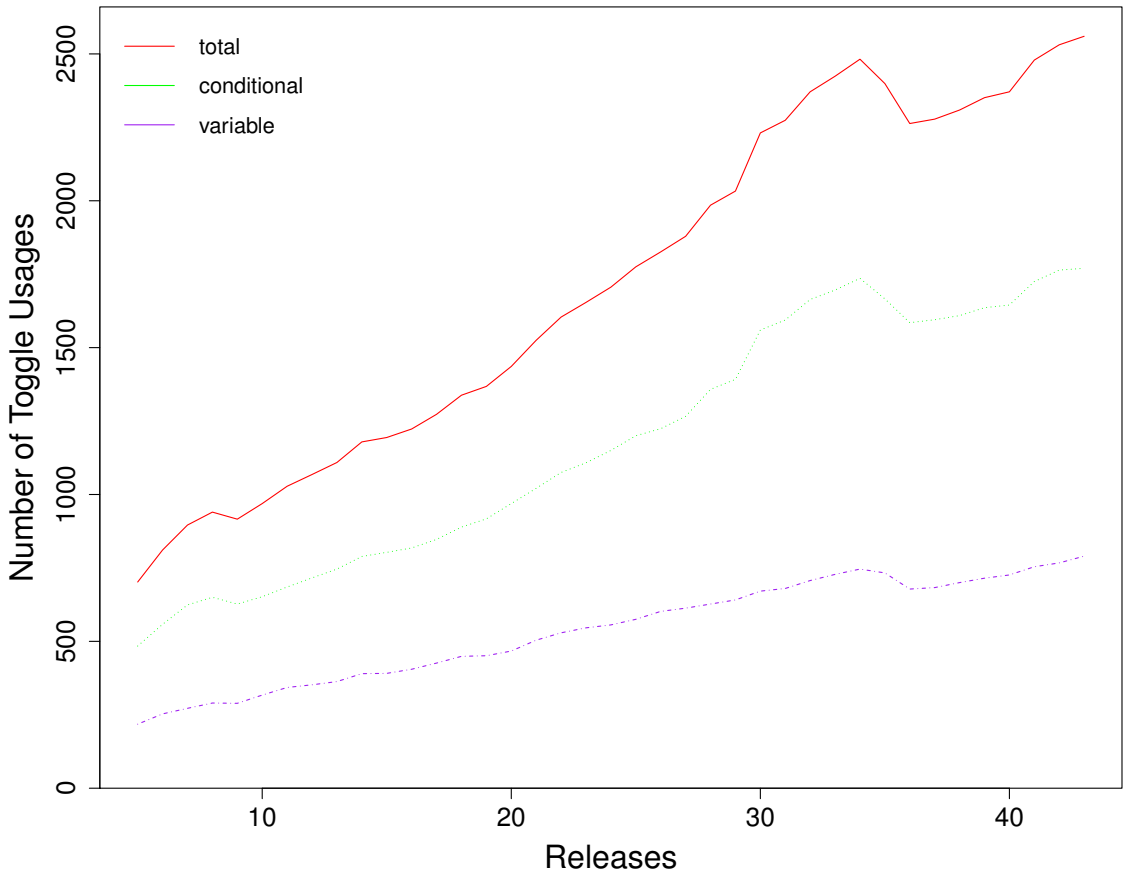


Figure 3.4: The total number of times toggles are used per release of Google Chrome as well as the type of usage: direct conditional or variable assignment.

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 [86].

```
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 3.5: 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.

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

Figure 3.6: Toggle return in chrome/content/renderer/render_process_impl.cc of release 29.

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

3.3.5 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 3.7 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

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

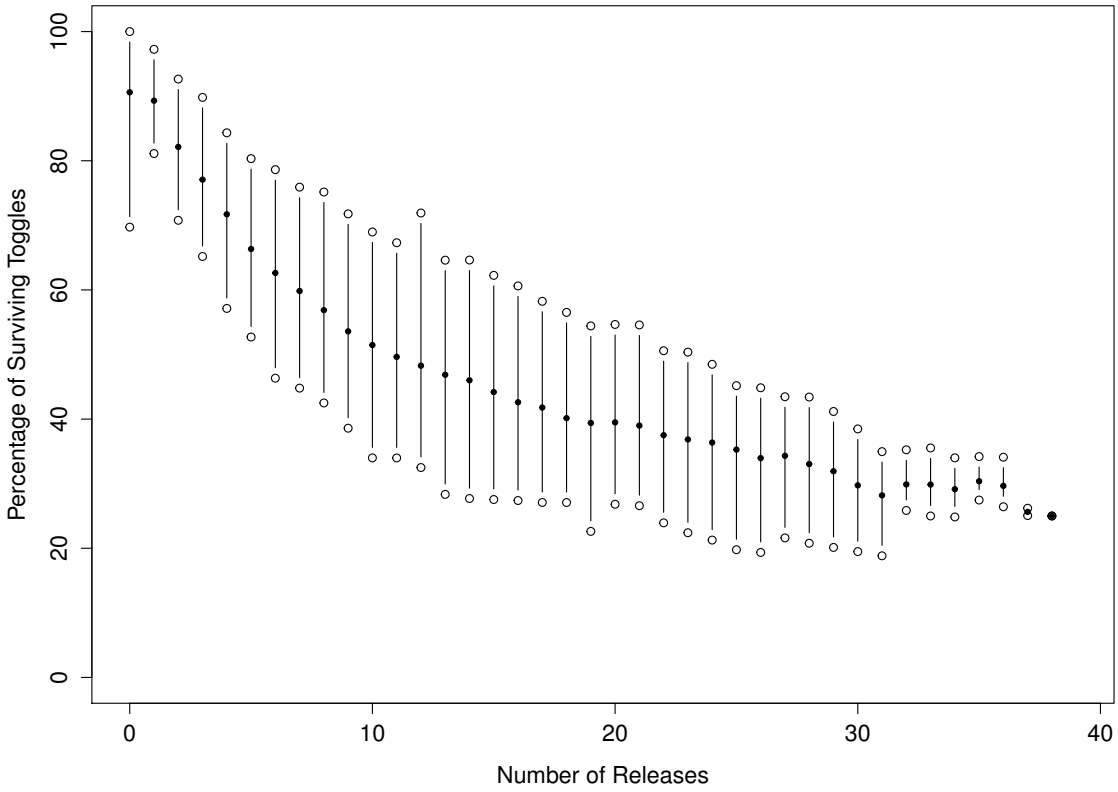


Figure 3.7: Survival curve showing the percentage of toggles that remain in the code base for 1, 2, . . . , 38 releases of Google Chrome. Each vertical bar spans between the 5th and 95th percentile, with the average as the middle point.

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).

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.

Table 3.1: Prevalence of each toggle type according to the toggles mentioned in the Chrome spreadsheet [74].

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

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

3.4 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 [74] created by Chrome developers in March 2014 (and still maintained at the time of writing this thesis).

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 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 3.8 shows the survival curves for the toggles of the three types identified in Table 3.1. 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.

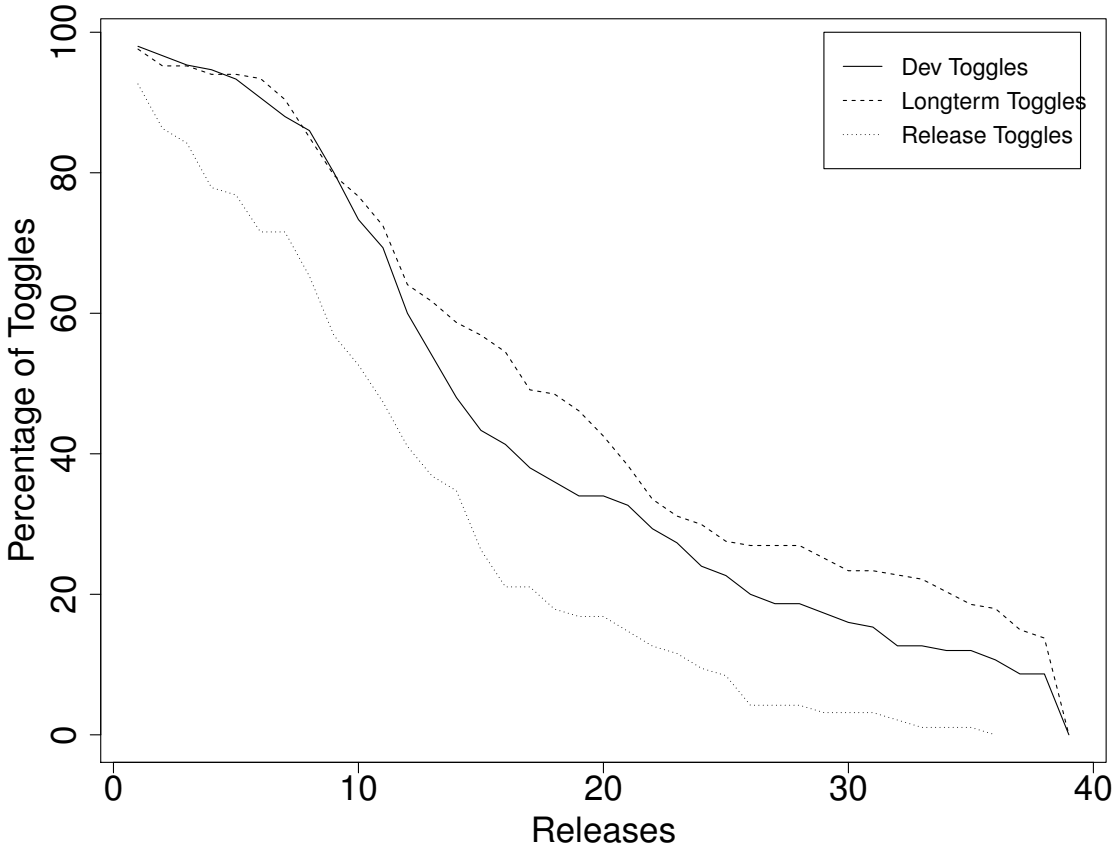


Figure 3.8: Survival curves for the three different toggle types, showing the percentage of toggles surviving 1, 2, etc. releases of Google Chrome.

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.

3.5 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 [2, 4, 17, 18, 22, 46, 54, 58, 64, 66, 67, 68, 86, 99, 114, 116, 117]. 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 [4], Facebook releases its website exactly twice a day [114] and Google releases a new major version of Chrome every 6 weeks [86]. 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 [69]. 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 [76].

For example, Laforge [86], 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 [68] and Facebook [114]). 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 [42] 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" [46], 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" [67]. 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 [111] (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 [86]. 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" [18]. 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% [10]. 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 [46]. 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. [76] 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 [68], 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 [117] 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 [114].

Developing new features behind a feature toggle requires discipline and additional effort during feature design and development [86]. 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" [46]. 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 [46]. 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 [17]. 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 [17, 46]. 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 [46]. 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" [17].

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" [105].

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 [46, 86]. 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 [114]. 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 [58]. 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.

3.6 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 [61]. 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 [74] 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 3.1 adds up to 724, which would correspond to the time frame around release 20 in Figure 3.4. 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.

3.7 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" [46]. 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" [17]. 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 [58]. 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.

Chapter 4

The Modular and Feature Toggle Architectures of Google Chrome

Note: this chapter has been submitted to Journal of Empirical Software Engineering (EmSE). A major revision has been performed and is presented in this chapter.

Abstract Software features often span multiple directories and conceptual modules making the extraction of feature architectures difficult. In this work, we extract a feature toggle architectural view and show how features span the conceptual, concrete, and reference architectures.

Feature toggles are simple conditional flags that allow developers to change features live in production. They are commonly used by large web companies, including Google, Netflix and Facebook to selectively enable and disable features.

We extract the feature toggles from the Google Chrome web browser to illustrate their use in understanding the architecture of a system. Since there is no overall conceptual and concrete architectures for Chrome, we had to manually derive them from available documentation and map them into the source code. These modular representations of a modern web browser allowed us to update the 12 year old browser reference architecture with current technologies and browser concepts.

Mining the usages of feature toggles in the source code, we were able to map them on to the modular representation to create a feature toggle architectural view of Chrome. We are also able to

show which features are contained in a module and which modules a feature spans. Throughout the paper, we show how the feature toggle view can give new perspectives into the feature architecture of a system.

4.1 Introduction

Software architecture plays a key role in software maintenance and evolution. Software engineers refer to the architecture in order to understand the software system before modifying or re-engineering it. Traditionally, the software architecture is derived from the existing documentation and the relation among source files, such as the call graph. For example, [19] architectural extraction process involves using the conceptual architecture diagrams and relations in the directory structure to reconstruct the concrete architecture.

Since traditional architectural representations are usually at the modular level, features are often difficult to represent as they can span multiple modules and are not explicitly delimited in the source code. For example, a feature as simple as logging into a system will *span* three layers requiring a user interface, a database with user names, and business logic to check that the password is correct for the user name.

Research into software product lines has focused on use case [62] and UML representation of features, however, it is difficult to project these separations of features into the code base and to keep the UML representation up-to-date [35, 12]. In industry, separate software product lines are often achieved by cloning code and modifying it to conform to the different feature requirements [115, 41]. This can lead to long-term maintenance issues as the two code bases evolve differently. It is also difficult to capture a single architectural diagram of the cloned code [41].

Recent developments at large web companies, such as Google and Facebook, have seen the need to isolate each feature so that the feature can be quickly turned off during botched releases and for feature comparison testing, such as A/B testing [39, 55]. To achieve this flexibility, many companies use feature toggles. The promises and perils of feature toggles were recently studied in a practitioner-focused work by Rahman *et al.* [109]. The goal of this work is to understand how feature toggles affect the architecture of a system.

4.1.1 What is a feature toggle?

Feature toggles are conceptually simple, they combine flags with conditionals to denote portions of the code that are related to a feature. Unlike compile time toggles (compile-time switches), such as *#ifdef*, feature toggles can change the behaviour of a running system. Our work contributes the feature toggle view which gives a new perspectives to the feature architecture of a system.

Furthermore, feature toggles do not require specialized frameworks or new programming languages. For example, below we see the conditional statements used to guard a block of C++ code that implements "*text input focus*" feature associated with the toggle *kEnableTextInputFocusManager* in Google Chrome. If the toggle is enabled, the method *IsTextInputFocusManagerEnabled* returns true and a manager is used to deal with text focus, otherwise, a simple text client is used. With feature toggles, all features are compiled and live with a toggle configuration file or database controlling which features are available to which users.

```
bool IsTextInputFocusManagerEnabled() {
 return CommandLine::ForCurrentProcess()->HasSwitch(
     switches::kEnableTextInputFocusManager
 );
}

...

if(switches::IsTextInputFocusManagerEnabled()){
    ui::TextInputFocusManager::GetInstance()
   ->FocusTextInputClient(
       &text_input_client
    );
}else{
    input_method->SetFocusedTextInputClient(
       &text_input_client
```

```
    );
}
```

To study the effect of toggles on system architecture requires the extraction of multiple architecture views across a large system. As a result, in this work, we have chosen to examine a single project in detail. We feel that Google Chrome is a representative project because Chrome uses feature toggles in the same way as toggles are used in other internal Google code bases [86]. Our detailed case study of Chrome allows us to understand and visualize how feature toggles affect the system architecture. We call this the feature toggle architectural view. Our scripts and data are available to researchers and practitioners who want to extract this new architectural view [108].

4.1.2 Extracted Architectural Representations

The major contributions in this paper are the extraction of four separate architectural representations: the conceptual architecture, the concrete architecture, the browser reference architecture, and the feature toggle architecture.

1. Conceptual Architecture: The conceptual architecture describes the major entities and relationships among those entities. Although there is extensive documentation for Chrome, we did not find a overall architectural representation. Examining this documentation, we derive a conceptual architecture for Chrome in Section 4.4.

2. Concrete Architecture: The concrete architecture is a modular architecture that is based on the source code. We manually mapped over 28k source files to their respective concepts to understand the modules present in Chrome. Using the call graph we assigned edges between these modules. We then compared the edges and modules found in the concrete architecture with the conceptual modules. We describe discrepancies in the documented conceptual modules and edges with those that actually exist in the code. See section 4.5.

3. Browser Reference Architecture: A reference architecture contains the components required for a particular system domain. In [63] Grosskurth and Godfrey extracted a browser reference architecture based on the Mozilla and Konqueror web browsers. We update this 12 year old reference architecture in Section 4.6.

4. Feature Toggle Architecture: We extract the *feature toggle architecture* of Chrome and

illustrate the new couplings and relationships that are revealed by this new perspective (Section 4.7). The feature toggle architecture can be viewed from a feature's or a module's perspective. From a feature's perspective the architecture shows which modules a feature spans, see Section 4.7.1. From a module's perspective the architecture shows which features are contained in a module, see Section 4.7.2. Since features necessarily span multiple modules, we discuss the new feature relationships among modules that the toggle architecture reveals. We also discuss the evolution of the feature toggle architecture across four releases in Section 4.8.

Our study shows that the feature toggle architecture can give new perspectives to the feature architecture of a system. We note that the feature toggle architectural does not replace other architectures or architecture extraction methods such as feature-oriented development or cross-cutting features in aspect-oriented development. In contrast, the goal of this work is to show how the widely-used practice of feature toggles allow for a new perspective on the architecture from the feature's perspective extracted from the source code. Our contribution is to make both researchers and practitioners working on projects with feature toggles aware of the impact of toggles on the system's architecture.

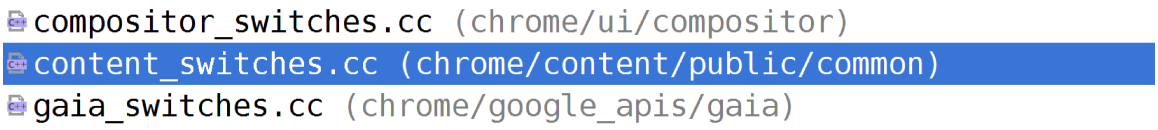
4.2 Background on Toggles and Google Chrome

Feature toggles are simple conditionals that guard blocks of code allowing developers to enable and disable features [109]. This mechanism facilitates dark launches [107], disabling features, and A/B testing during rapid releases [3]. From a practical perspective, features require the following [58]:

(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 start-up 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.



(a)

```
107  // Disable 3D inside of flapper.
108  const char kDisableFlash3d[]             = "disable-flash-3d";
109
110  // Disable using 3D to present fullscreen flash
111  const char kDisableFlashFullscreen3d[]   = "disable-flash-fullscreen-3d";
```

(b)

```
529  CommandLine* command_line = CommandLine::ForCurrentProcess();
530  if (command_line->HasSwitch(switches::kDisableFlashFullscreen3d))
531    return;
532  WebKit::WebGraphicsContext3D::Attributes attributes;
533  attributes.depth = false;
534  attributes.stencil = false;
535  attributes.antialias = false;
536  attributes.shareResources = false;
537  context_ = WebGraphicsContext3DCommandBufferImpl::CreateViewContext(
538      RenderThreadImpl::current(),
539      surface_id(),
540      NULL,
541      attributes,
542      true /* bind generates resources */,
543      active_url_,
544      content::CAUSE_FOR_GPU_LAUNCH_RENDERWIDGETFULLSCREENPEPPER_CREATECONTEXT);
545  if (!context_)
546    return;
```

(c)

Figure 4.1: Examples of Chrome toggles: (a) the sets of toggles composing a related feature set contained in "switch" files, (b) setting the toggle value inside `content_switches.cc`, and (c) code that is covered by the toggle *kDisableFlashFullscreen3d*.

Toggles have been used in Chrome since release 5.0 in 2010. Instead of having a single configuration file for feature toggles, Chrome has multiple "switch files" containing toggles for a relevant set of features *i.e.* "feature set" (see Figure 4.1a). For example, there is a feature set that contains toggles to manage "Content" related features (`content_switches.cc`) which includes the feature toggle *kDisableFlashFullscreen3d*.

In Chrome, feature toggles are string constants that can be set or unset. For example, Figure 4.1b 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 [20]. These feature toggles are then used inside

conditional statements, as shown in Figure 4.1c. If the toggle *kDisableFlashFullscreen3d* has been set a `return` statement exits the method without executing the 3D rendering feature.

4.2.1 Chrome Data

To study the feature toggle architectural view, we collect the feature toggles for 30 consecutive release versions that cover more than five years of the development history of Google Chrome to understand the use of toggles on the system overtime. We study four separate release version: 5.0, 13.0, 22.0, and 34.0. We selected these releases because they cover the entire observing time period keeping an almost equal gap between each of the release versions. The versions contain 7K, 11K, 17K, and 28K C/C++ files and 1.0M, 1.4M, 2.2M and 3.7M lines of code respectively (Table 4.1). We removed the third-party code in the Chrome repository and external plugins before conducting our analysis.

Table 4.1: Google Chrome source code versions

Rel. Date	**#Dirs**	**#Files c, cc**	**#Feature Sets**	**#Toggles**	**Size MLOC**	**#Files Using Toggles**	**%Files Using Toggles**
5.0 May 21, 2010	1,455	7,213	6	272	1.0	4,550	63.08
13.0 Aug 2, 2011	2,206	10,886	11	466	1.4	4,992	45.85
22.0 Jul 31, 2012	3,400	17,144	21	726	2.2	10,927	63.73
34.0 Apr 8, 2014	5,411	28,585	33	1,031	3.7	12,545	43.88

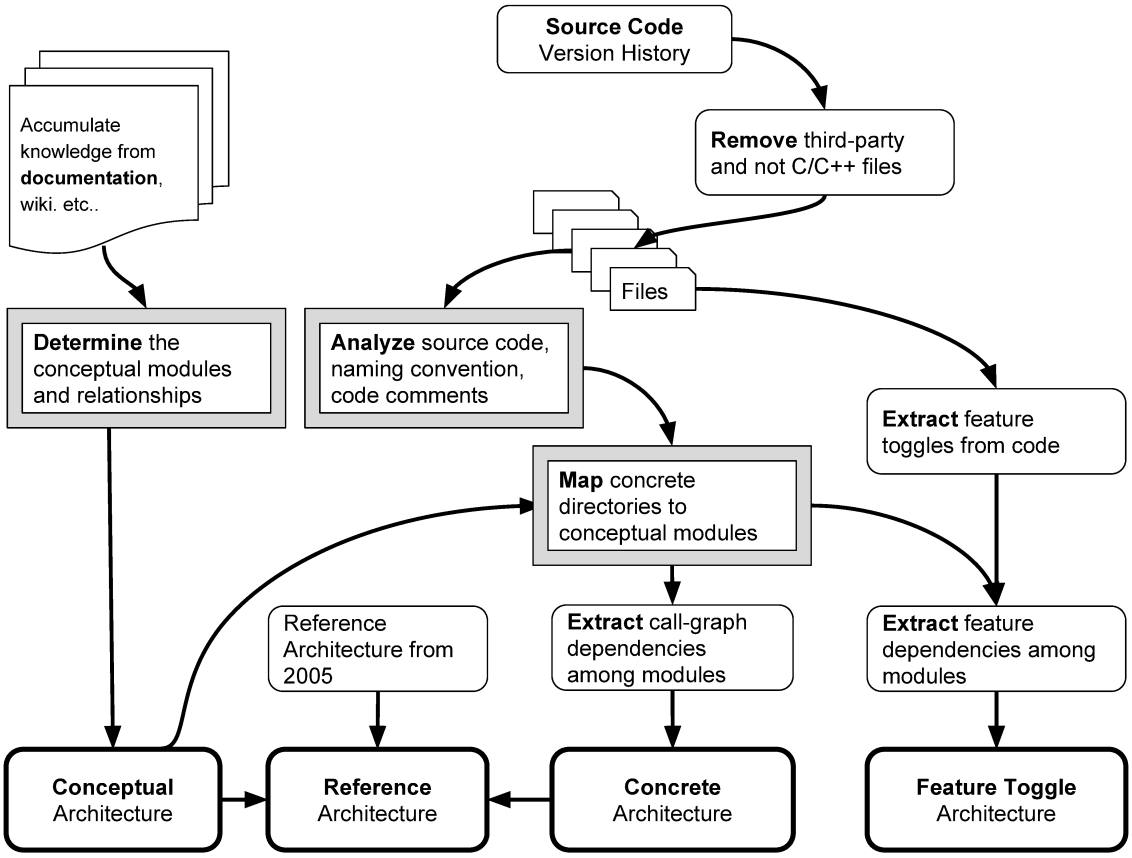


Figure 4.2: Extraction steps for conceptual, concrete, reference, and feature toggle architectural extraction. Square double boxes represent steps that required substain human input, while curved boxes represent semi-automated steps.

4.3 Architectural Extraction Process

In Figure 4.2, we show the architectural extraction stages for the four architectures: the conceptual, concrete, reference, and feature toggle architecture.

4.3.1 Conceptual Architecture

Google Chrome does not have an existing overall conceptual diagram describing the relationships between entities in the system. We derive the conceptual architecture from the existing documentation for Google Chrome including the web resources and official wiki [24],[31],[25],[26], [27],[28],[29]. During a course on software architecture, a team of four graduate student, lead by the lead author and supervised by the third author, manually read and discussed the role of the each

modular entity and the relationships among them. Understanding the documentation and creating an initial diagram took the team 2 person-weeks. The resulting architectural diagram is Figure 4.3 in Section 4.4.

4.3.2 Concrete Architecture

Table 4.2: Sample of file to directory and directory to conceptual module mappings

File	Directory	Conceptual Module
`shelf_widget.cc`	`ash\shelf`	`Browser View`
`tabs_api.cc`	`chrome\...\extensions\tabs`	`Extensions`
`avatar_menu.cc`	`chrome\...\profiles`	`Data Persistence`

The concrete architecture is based on the source code files. We map the directories to the conceptual modules and then extract the call graph to the relationships between the modules. The steps are as follows:

(1) **Map concrete directories to conceptual modules:**

Following [19] technique, we extract the source code and map the files and directories to modules based on the modular entities extracted from the documentation in the conceptual architectural extraction stage. Each source file is examined for its purpose based on its location in the system, name and naming conventions, source code comments, and any documented knowledge.

This manual mapping was performed by a group of four graduate students during a course on architectural extraction. The the first author lead the team and the third author supervised the project. For release 5.0, 13.0, 22.0, and 34.0 the students examined 7.2k, 10.8k, 17k, 28.6k, respectively. In total the mapping took 4 person-weeks.

The output of this stage is a mapping between conceptual modules and directories. Table 4.2 provides an example of these mappings with the the full set of mapping available in the replication package [108].

(2) **Extract the call graph dependencies among modules:**

For each of the source files, we extract the call graph using the "Understand" tool. Combining the call graph and and the module mapping list, we obtain the relationships between modules.

(3) **Construct the concrete architecture:**

Instead of using *lsedit* like [19], we use an online graph editing tool [34] that generates directed graphs from the module-module relations. We then, normalize the inter-module relations to construct a readable concrete architecture. We removed the modules Widget, Skia, Installer, Tools, Downloads, Services, Test, Debug and Remote Host, as they are not core to the functioning of the browser. We do not show the edges for the `Base` and `Utility` modules as more than 46K calls to `Base` made by 48 modules and 2.8K calls to `Utility` made by 20 modules. We label the edges with the number of calls. Figure 4.4 in Section 4.5 shows the concrete architecture for Chrome version 34. The full list of call dependencies among modules is available in our replication package [108].

4.3.3 Browser Reference Architecture

A reference architecture contains the fundamental entities and relationships among them for a particular domain. It serves as a template to understand the system architecture in a domain. We update the 12 year old browser reference architecture that was proposed by [63] with the extracted modern conceptual and concrete architectures of Google Chrome. New modules are added based on technology and design changes. We also verify that conceptually these modules have been adopted by other browsers, including Firefox and Safari. Figure 4.5 in Section 4.6 shows the browser reference architecture.

4.3.4 Feature Toggle Architecture

The feature toggle architecture represents the feature toggle dependencies among the modules. Feature toggles are embedded in the source code files. Once we identify the feature toggles, we can assign feature dependencies among concrete and conceptual modules. The steps are shown on the right side of Figure 4.2 and discussed below.

(1) **Extract feature toggles from code:**

Table 4.3: Sample of module to feature set mappings

Module	Feature Set
Browser View	chrome_switches
Browser View	ui_base_switches
Browser View	ash_switches
Content	base_switches
Content	content_switches
Content	ipc_switches
Content	ui_base_switches
Content	shell_switches
Render Engine	chrome_switches
Render Engine	gpu_switches
Render Engine	gl_switches

We extract all the feature sets ("*_switches.cc*" files) and the toggles within those files (for example, *kDisableTranslate* in translate_switches.cc) from the source code. We drop all the files that do not dependent on any feature toggle and keep only the files that use at least one feature toggle.

(2) **Extract feature dependencies among modules:**

Using the directory to module mapping developed for the concrete architecture we can we know which file belongs to which module. Modules that contain feature toggles from the same feature set, have a feature dependency among them. Similarly, from the features perspective we can see which modules a feature spans. We generate two mapping lists for module-feature and feature-module dependencies as the examples shown in Table 4.4 and 4.3. The full list of feature and module mappings can be found in our replication package [108].

(3) **Construct feature toggle architecture:**

Using the graph editing tool [34] we generate the graphical representation of the dependencies between modules and features, see Figure 4.6 in section 4.7.

4.4 Conceptual Architecture

The conceptual architecture is derived from the Google Chrome development documentation and describes the conceptual entities and relationships in the browser. The architectural extraction

Table 4.4: Sample of feature set to module mappings

Feature Set	Module
chrome_switches	Render Engine
chrome_switches	Browser Engine
chrome_switches	Browser View
chrome_switches	Apps
chrome_switches	Data Persistence
chrome_switches	UI
chrome_switches	Network
chrome_switches	Chrome OS
chrome_switches	Print
ui_base_switches	Browser View
ui_base_switches	Content
ui_base_switches	Browser Engine
ui_base_switches	Utility
ash_switches	Browser View
ash_switches	ASH
ash_switches	UI

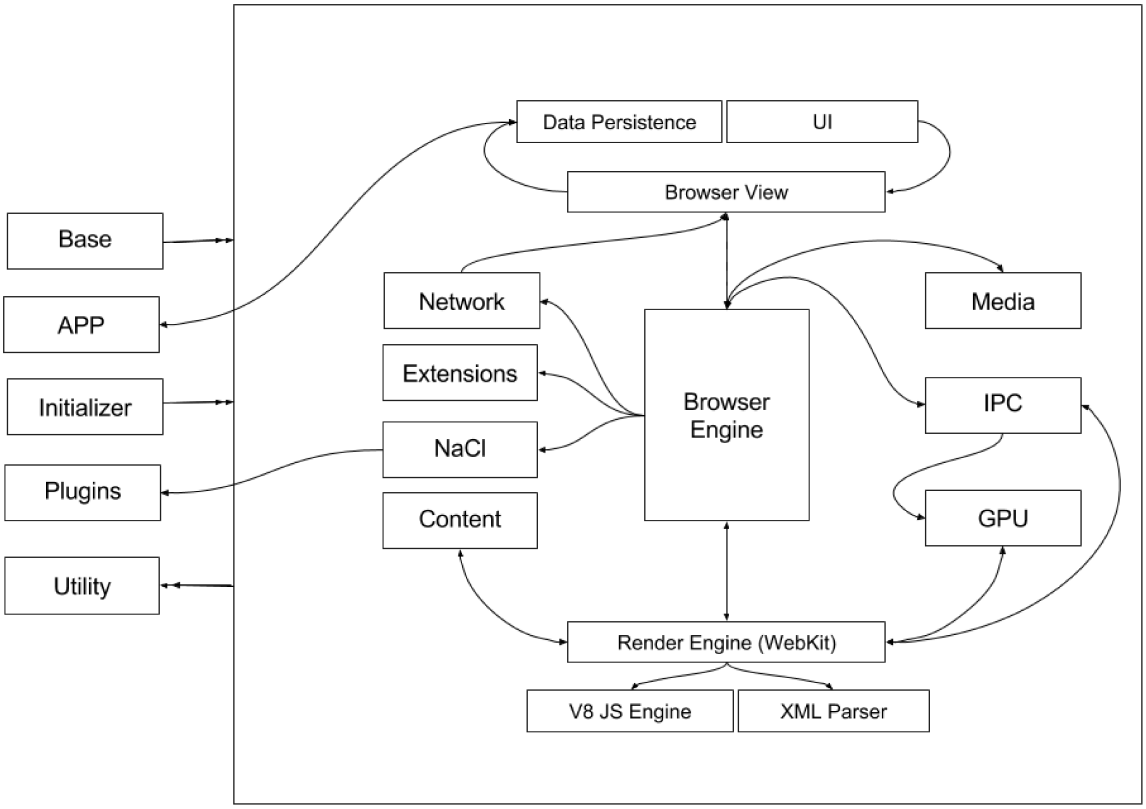


Figure 4.3: Conceptual architecture of Google Chrome.

process we followed is described in Section 4.3.1. Figure 4.3 shows the conceptual architecture of Google Chrome. In this figure, the solid square boxes represent the modules and the edges indicate the dependencies between them. The "`Utility`", "`Base`" and "`Initializer`" modules have relationships with all modules. From the Chrome developer resources we identified 19 conceptual modules and 21 edges between those modules. We discuss the major modules below:

The **UI** module is the front end of the Google Chrome web browser that displays the rendered HTML with CSS, Text, Videos, and Images. The `UI` also provides developer tools, including event statistics for a page and the ability to inspect the HTML and Javascript elements.

The **Browser View** or UI Backend controls the the views that makeup the `UI` module, including the tabs, toolbar, and bookmarks bar. It represents these views in the DOM tree. The Browser View collects the rendered contents from the `Render Engine` to create the DOM tree.

The **Data Persistence** module stores data on the user's local machine to create a more efficient browsing experience. For example, browsing history, auto-fill, passwords, user preferences are stored locally so that they can be quickly accessed by the views in `Browser View`.

The three modules discussed above handle the display part of the browser. While the main processing of the web page information is done by the `Browser Engine` and `Render Engine` with the help of other modules such as `Content`, `GPU`, `NaCl`.

The **Browser Engine** is a communicator between `Browser View` and the `Render Engine`. It is the core of the browser which bridges the user interface and rendering engine. It initiates all web page events, including loading, form submission, reloading, and navigation.

The **Render Engine** renders the web page from a URL. It interprets marked up content (such as HTML, XML, and image files) and formatting information (such as CSS, XSL) with the help of the `Layout Engine`. It shares the graphic data with `GPU` to be processed separately. The `Render Engine` uses the **V8 JavaScript Engine** and **XML Parser** to process Javascript and XML contents.

The **Content** module implements the core Chrome rendering features. It also contains factory classes to create the necessary WebKit objects for Chrome. The goal of this module is to clearly separate the rendering code from other browser features.

The **GPU** provides secured access to the system's 3D Graphic APIs [[25]] and accelerates the

processing of the graphical contents. The Render Engine module uses GPU to process and render the graphic elements in a web page.

The **IPC** module allows each tab to run as its own process facilitating interprocess communication with the core browser.

The **Extensions** module serves as a point to plug-in third-party features to enhance browser functionality. Extensions are managed by the Browser Engine [29].

NaCl is a sandbox that allows Plugins to run across platforms without being recompiled. It also improves the security and portability of the native Chrome code [27].

The **Network** module interfaces with the hardware to allow network connections.

4.5 Concrete Modular Architecture

The concrete architecture is derived from manually mapping the source code directories to the conceptual modules. Since there are over 5.4K directories in Chrome release 34.0, a team of 4 graduate students took 4 person-weeks to manually map each directory to a corresponding conceptual module. After this mapping is complete, the call graph is extracted using Understand to create the relationships between modules. Finally, the concrete diagram is drawn using a graph editing tool to improve readability. The details of the architectural extraction process is described in Section 4.3.2.

The concrete architecture is shown in Figure 4.4. We identified five additional modules, which were not present in the conceptual architecture: Printing, Cloud Printing, Component, PPAPI and Base View. None of these modules were explicitly discussed as separate entities in the documentation; however, when we examined the source code, the files and related directories were separate from the directories that mapped to the conceptual modules. We briefly describe the rationale for adding these additional modules. The Printing and Cloud Printing modules clearly have separate directories, but are not discussed in the documentation. The Component module implements the components that are then used in the Browser View, but resides outside of the concrete Browser View directory. Although conceptually the flash plugin belongs in the Plugins component, the implementation of this plugin, PPAPI, is in its own directory and is tightly coupled to other parts of the system. The Base View was re-factored out of the Base

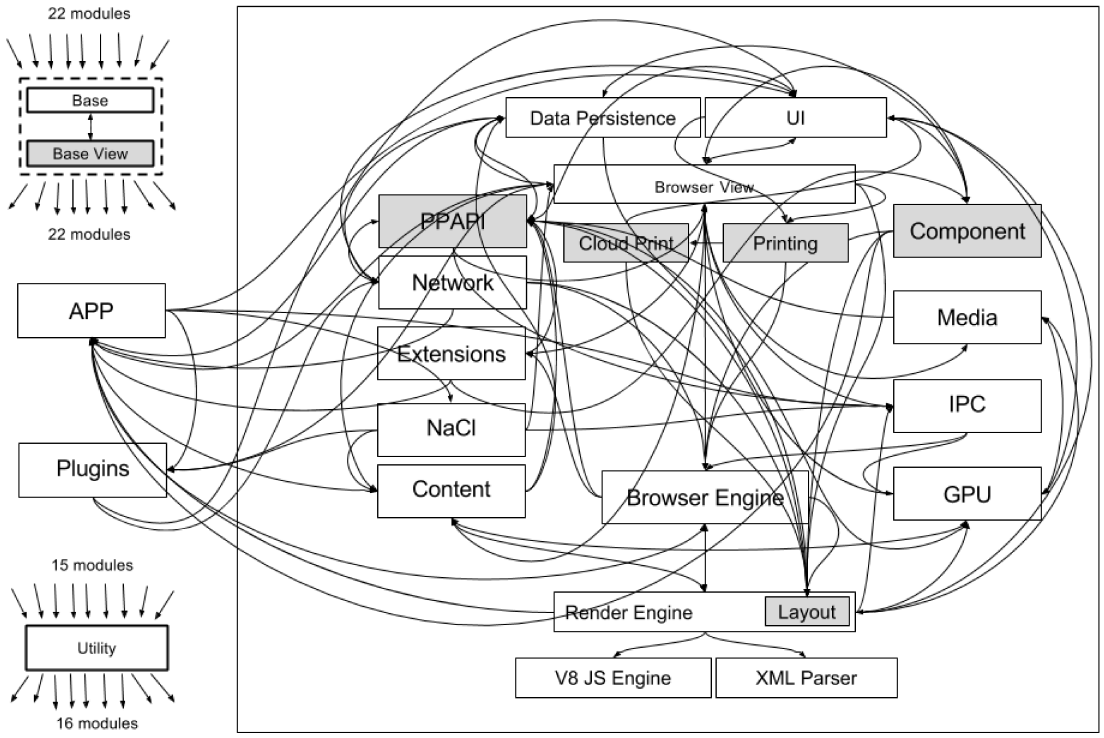


Figure 4.4: Concrete architecture for Chrome Release 34.0. The gray edges represent relationships between modules that did not exist in the conceptual architecture.

module in Chrome version 13.0 to delimit the view components in the base.

The concrete architecture has 71 edges compared to the 21 edges found in the conceptual architecture. The full list of edges and the number of total calls can be found in our replication package [108]. Since there are 53 new edges, we cannot discuss them all. Instead, we discuss some interesting architectural violations that we discovered in Chrome.

For example, the `Network` module has five new dependencies with `Layout`, `PPAPI`, `UI`, `App` and `Plugins`. When we examined these unexpected dependencies, we discovered that developers avoided existing interfaces. Instead, they called methods from the `Network` module directly.

The conceptual architecture contains an edge between the `GPU` and `NaCl` modules, which is missing from the concrete architecture. On investigation we found that the `GPU` receives the serialized system calls from `NaCl` via a ring buffer in shared memory. The `GPU` then parses the serialized commands and executes them with appropriate graphics calls.

A similar shared memory strategy is described in the documentation whereby the `GPU` output is communicated through the `Render Engine` on the shared memory to the `Browser Engine` and `Browser View` to finally be displayed in the `UI`. However, we observe direct calls between `GPU` and the `Browser View`, and `UI` showing an unexpected direct coupling between these components.

The `Content` module exposes an API for the core rendering code. In the documentation it is unclear which modules actually use the `Content` module. The Render and Layout Engine conceptually need to call the `Content` module, however, we in the figure that there are six other modules calling the Content module.

The `Base` and `Utility` modules that contain library and other generally reusable functionality which have call relationships with 22 and 16 modules, respectively.

[19] examined some of the unexpected calls found in the Linux kernel's concrete architecture. Like us they found that developers sometimes avoided interfaces to make the system more efficient. We can see this in the case of `Layout` module that bypasses interfaces in the `Browser View` to directly access `UI`.

Unlike [19] who found no additional modules in the Linux concrete architecture, we found that as Chrome evolved some modules and relationships that conceptually should not exist still remain

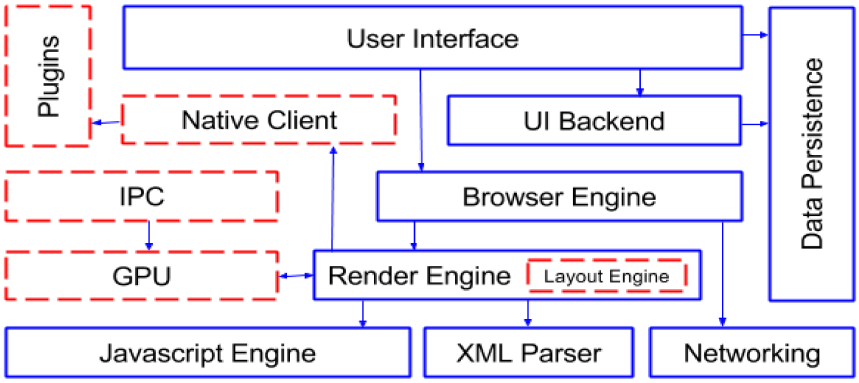


Figure 4.5: Modern browser reference architecture. Boxes with red dashed borders are new components not present in the [63] browser reference architecture.

as awkward legacies. For example, Chrome is replacing the Netscape flash plugins, NPAPI, with their own Plugin (PPAPI). As the migration continues, the NPAPI continues to be in the source code but is flagged as an obsolete module [30].

4.6 Browser Reference Architecture

From our extracted extracted conceptual and concrete architectures of Chrome, we update the 12 year old browser reference architecture that was proposed by [63]. Figure 4.5 shows the reference architecture. The solid boxes are the modules that are present in the 2005 reference architecture. The red dashed boxes are modules that are new to the modern browser, including the `GPU`, `IPC`, `Plugins`, `NaCl` and `Layout Engine` modules. We discuss why each new module was included referring to other modern browsers. The extraction process is detailed in Section 4.3.3.

Graphical Processing Units (GPUs) have become increasingly more common. Modern web browsers, including Chrome, Firefox, Internet Explorer, Safari have the hardware acceleration module (GPU) on by default. The GPU reduces CPU consumption significantly which increases the overall performance of the of the web browser.

Multi-processor architectures have lead to a greater need for communication among asynchronous processes. Furthermore, by isolating each browser tab in a separate process problems encountered in a single tab are less likely to affect other tabs. Chrome has an Inter-Process Communication

(IPC) module responsible for coordinating messages passed between process. [48] and other popular modern web browsers adopted a similar multi-process architecture for efficient browsing.

Extending the functionality of applications with plugins has become an essential requirement of adaptable software ecosystems. Chrome has a `Plugins` module to create a clean interface for other developers to add functionality. For example, the "Microsoft Office" plugin developed by the developers at Microsoft allows Chrome users to access documents created by Microsoft Office through the Chrome browser. Chrome is not alone, all the modern web browsers can plug-in different plugins provided by other software systems, for example, Firefox has many plugins ranging from an *iTunes adapter* to *fire-bug*, *paint* and *screen capture*.

Web applications that require high performance across platforms can take advantage of the NAtive CLient `NaCl` sandbox in Chrome. This module allows the compiled C/C++ code to run with more low-level control while maintaining security and efficiently across platforms [123]. `NaCl` is especially useful for 3D games, CAD modelling, client-side data analytics [123]. Firefox has the similar native functionality provided by the module "*OdinMonkey*" [101].

Traditionally the `Render Engine` took the response from the `Browser Engine` and generated the tree of the HTML elements. In Chrome the task of calculating the HTML element position has been moved into a separate sub-module called the `Layout Engine`. Not only Google Chrome but also Firefox introduced the Layout Engine to parse the HTML and create the frame tree for the DOM [47].

The intervening 12 years since [63] browser reference architecture has seen significant technological changes, such as the introduction of GPUs. While many of the conceptual modules remain the same, both external and internal conceptual boundaries have increasingly become more defined. Our updated reference architecture provides a new guide for developers seeking a conceptual understanding of the modern web browser.

4.7 Feature Toggle Architecture

The use of feature toggles in the source code allows us to extract an architectural view of the features in Chrome. This helps us to identify new dependencies among modules and to better

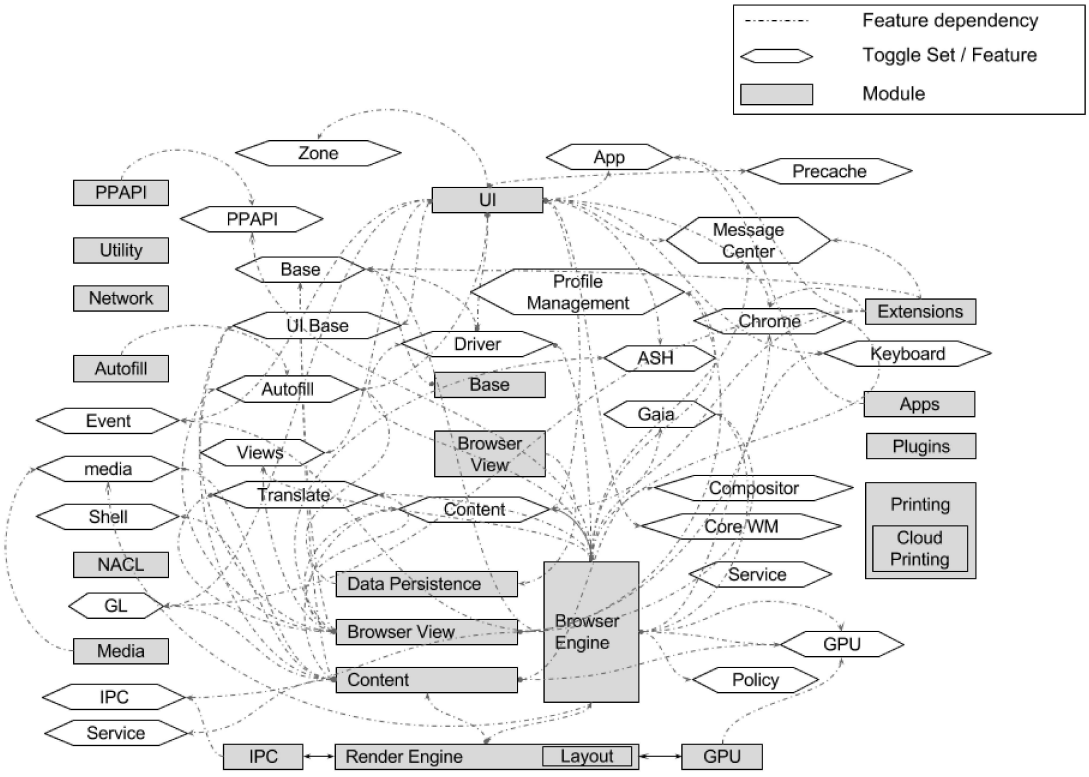


Figure 4.6: Feature toggle architecture

understand which modules a feature spans and which features are contained in a module. The feature architectural view helps a developer working on a feature to determine which modules will be affected by implementing a new feature or a change to an existing feature. Similarly, a developer maintaining a module will know which features he or she needs to be familiar with.

Since feature toggles are used in source code files, we can use the directory to conceptual module mapping derived from the concrete architecture to extract the feature toggles that affect each module. The details of the extraction process can be found in Section 4.3.4

Figure 4.6 shows the feature toggle view of the Chrome version 34. The square boxes are modules and the hexagonal boxes are named sets of features. For example, the hexagonal box "Translate" contains the feature toggles for the features related to translation, such as *kTranslate-ScriptURL*. Relationship edges between modules are drawn when two modules use the same feature toggle. There are too many edges to show in the figure, so we exclude edges that have less than 7 common feature toggles. The full list can be found in our replication package [108].

The feature toggle architecture contains 1,173 edges. Compared to the concrete architecture that has 71 edges, 1,102 are new. Compared to the conceptual architecture that has 18 edges, 1,155 are new. These new edges represent natural feature dependencies which are not obvious in the traditional conceptual and concrete architectures. We discuss the feature architecture from the feature's perspective in the next and from the module's perspective in Section 4.7.2.

4.7.1 Feature's Perspective of the Feature Toggle Architecture

Features necessarily span multiple modules. For example, a "Password" feature will span at least two modules as a password must be entered in the `UI` and persisted in a data store. The feature perspective allows a developer to be aware of the modules that a feature spans. To quantify feature span, we plot the distribution of the number of modules features span in Figure 4.7. A box-plot also shows the quantiles of the distribution with the line representing the median. In the median case, a feature will span between three to four modules. This span implies that a developer will have to have an understanding of around four modules to fully understand a feature. Furthermore, certain features are highly complex spanning up to a maximum of 19 modules (*e.g.,* base features). Since there are more than a thousand feature toggles, we provide some illustrative examples.

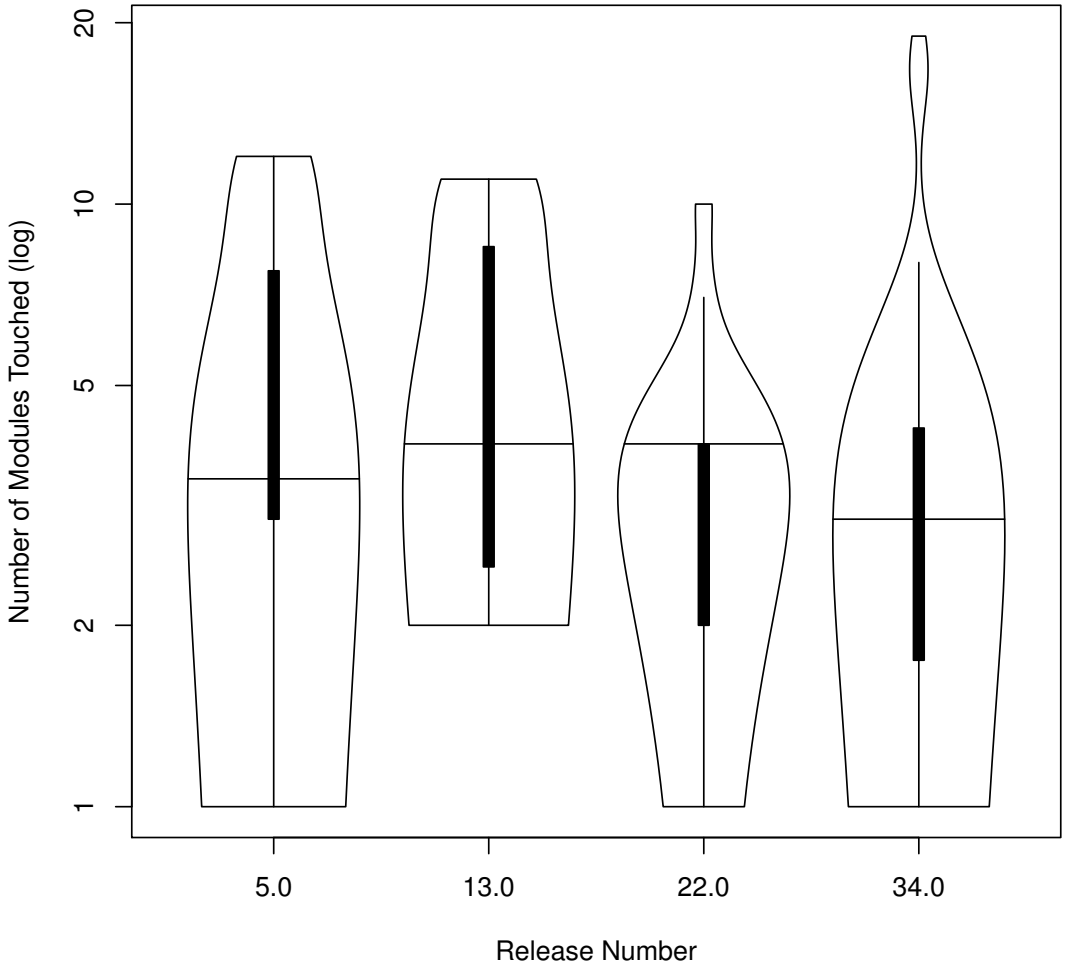


Figure 4.7: The distribution of the number of modules spanned by each feature

The first example is the *kEnableAutoFill* which allows user information to be cached. This feature toggle spans multiple modules including the `UI`, `Browser View`, and `Data Persistence` modules to auto-populate the password, user name, and other frequent user data. This does not violate or contradict with conceptual or concrete architecture because conceptually `Browser View` has a relation with `Data Persistence` to persist user data from UI. However, from the feature view, we understand if a developer has to work on a feature such as storing password, he or she will need to touch these three modules.

The second example is the VSync feature that controls video acceleration and prevents frame-rate "stuttering" and "screen tearing" in videos. This feature is provided by the graphics library (GL) and developed under the toggle "*kEnableGpuVsync* where GPU graphic acceleration is needed. This toggle is used in `Render Engine`, `Browser Engine`, and `UI` modules to toggle code that conditionally allows the VSync feature to be enabled. Although this feature spans multiple modules, the feature dependencies are not surprising because the concrete architecture has call relationships between the `GPU` module and the `Render Engine`, `Browser Engine`, and `UI` modules. However, the relationships between `UI` and `GPU` as well as `Browser Engine` and `GPU` are unexpected with respect to the conceptual architecture.

The third example is the Process Channel feature which is developed and controlled by the toggle "*kProcessChannelID*" that informs the child processes of the channel to listen on. As soon as a child process is spawned asynchronously by the `Browser Engine`, `Render Engine` or `GPU` modules the process channel is communicated to the `NaCl`, `App`, `Utility` and `Cloud Printing` modules. The implementation of this feature in multiple modules seems consistent with the concrete architecture as we see there are relationships between `IPC` and the modules that this feature spans. However, there are some new relationships when we consider the conceptual architecture, including those between the `IPC` and `NaCl` modules and the `Render Engine` and `Browser View` modules.

We have shown through examples, that the feature toggle architecture can allow a developer to identify the code base related to a feature. Although impact analysis may be used, our view on toggles provides the concise locations that are affected when making feature modifications.

4.7.2 Module's Perspective of the Feature Toggle Architecture

A developer should be aware of the features that are contained in the module in which he or she is working. We first quantify the number of features contained in each module. The distribution of features per module is shown in Figure 4.8. The figure also contains a boxplot imposed inside the distribution with the quantiles and a line at the median. In the median case a developer will have to understand between one to two feature sets per module. Compared with the overwhelming number of edges in the entire feature toggle architecture, we observe that a typical module contains a manageable number of features. However, we can see that over time features are being added to modules. For example, in release 34.0 the 75th percentile is at five features per module and there is one module that contains 59 features from 17 different feature sets. These modules are likely more difficult to understand and maintain. Chrome's size does not allow us to discuss each module in detail as a result we take three examples of important modules and associated features.

The first example is the `UI` module which is responsible for interacting with the end user by providing features such as keyboard interactions and automatic form filling with user information. Figure 4.9 shows that the `UI` module has 17 feature sets and contains 59 feature toggles in total. The hexagonal boxes represent the feature sets while the caption boxes provide two illustrations of the feature toggles. The full list of feature toggles is available in our replication package [108]. As an example, the "Event" feature set contains *kTouchEvents* which enables the touch based events. Another example, is the toggle *kCloudPrintFile* of the Chrome feature set that displays the cloud print dialogue and allows upload of files for printing.

Furthermore, the `UI` module touches many features because it is simple to disable access to the defective or incompatible features by removing the interface to the feature. For example, when the transparent "Layered Window" (also known as glass window) feature was released for Chrome browser, developers found it incompatible with the Direct3D surfaces. As a result, developers disabled this UI feature by using a release toggle *kGlassFrameOverlayAlpha*. Although the toggle architecture shows that `UI` module contains a large number of features, the conceptual architecture does not reflect this high coupling.

A second example is the `Browser Engine` which is responsible for rendering the HTML and

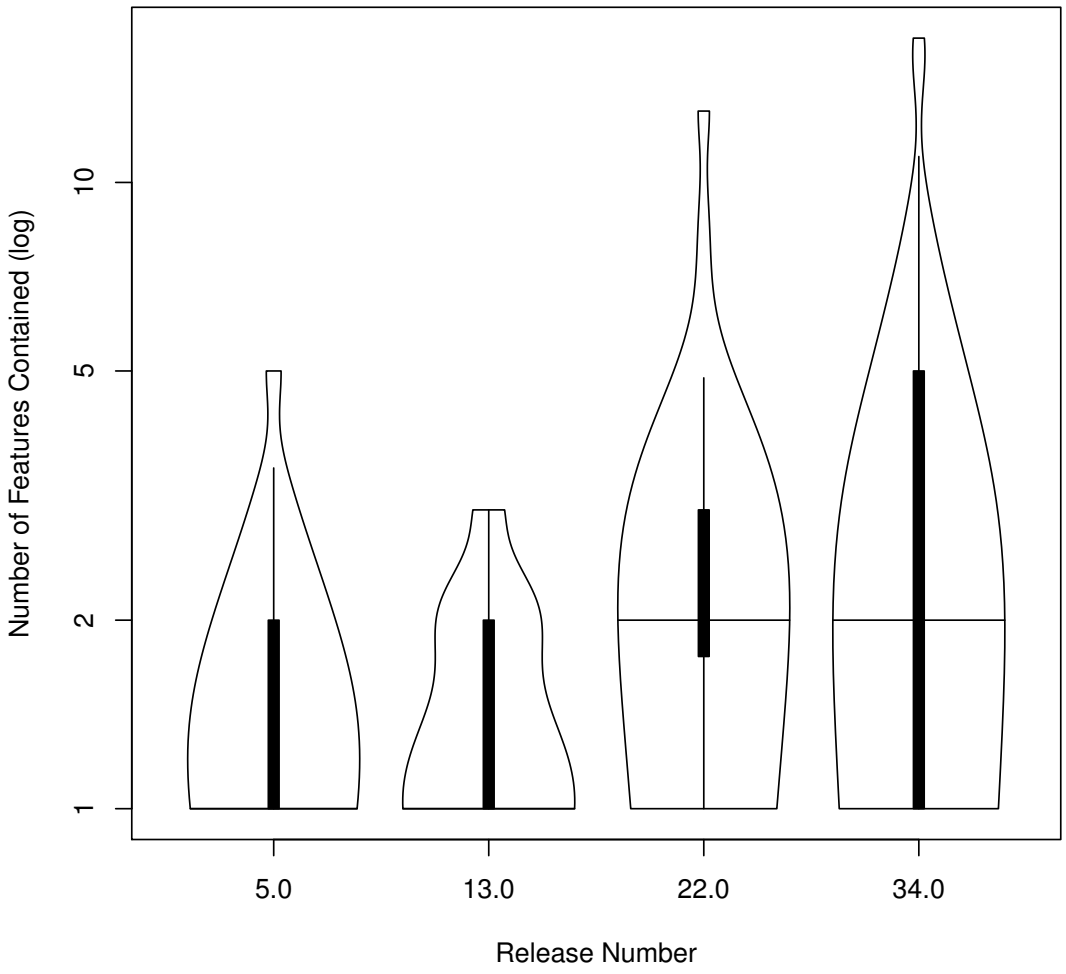


Figure 4.8: The distribution of the number of features contained per module

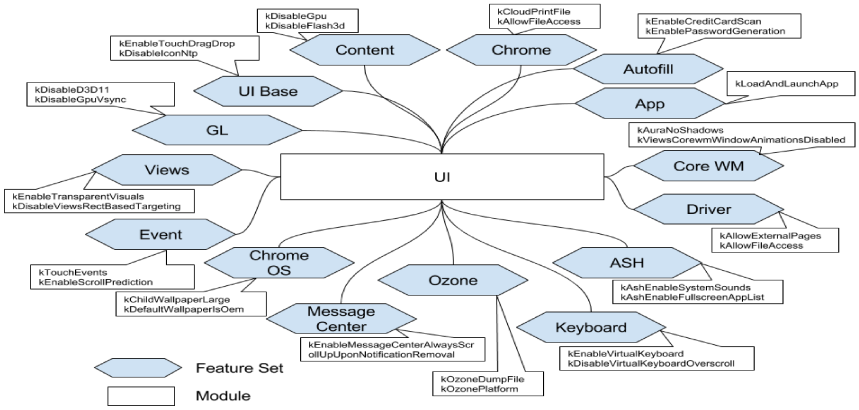


Figure 4.9: UI Module containing features - Chrome Release 34.0

CSS into the properly formatted visual objects. The `Browser Engine` contains the largest number of features of any module. One of the feature toggles is the *kEnableGpuShaderDiskCache* which allows the `Browser Engine` to receive rendered output from `GPU` and `Render Engine` via the shared cache memory. `Browser Engine` also contains features such as "Virtual Keyboard", "Translate" that are controlled by *kEnableVirtualKeyboard* and *kDisableTranslate* toggles respectively. Since the `Browser Engine` is one of the core modules, it is not surprising that this module contains a multitude of features that span from other modules. We do note that some features cause violations of the conceptual architecture. For example, the feature toggles *kEnableGpuAcceleration* has been used in `Browser Engine` to control code that depends on the GPU acceleration feature. However, we do not see any direct relationship between `GPU` and `Browser Engine` in the conceptual architecture.

The third example is the `Plugins` module which provides a consistent interface for external plugin code that does not depend on the native Chrome functionality. As a simple plugin point, there are few features toggles. An example toggle is *kEnablePluginPowerSaver* which globally informs plugins to use a power saving strategy.

The examples illustrate the complex features that make up Chrome's modules and feature toggle architectural makes these feature dependencies relationships clear to developers and researchers.

4.8 The Effect of Feature Toggles on Architectural Evolution

As a system evolves, features and modules can become coupled leading to violations of the modular architecture. From a raw size perspective, Chrome has grown substantially. In release version 5.0 Chrome had a total of 1.4K directories and 7.2K C/C++ files. Relative to Release 5.0, releases 13.0, 22.0, and 34.0, had an increase of 3K, 7K and 11K files, respectively. At release 34.0 there are 5.4K directories and 28.5K files.

Relative to the file size increases, the number of toggles was initially growing at a similar rate. However, over the releases the rate of increase in number of toggles is slower than the rate of increase in number of files. In early releases of Google Chrome, there were very few feature toggles in use. As the project grew, the number of feature toggles grew and they spanned more modules. At release 34.0, Chrome developers initiated a cleanup of unused toggles and associated dead code [[109]], which resulted in a drop in the number of total toggles and reduced feature span across modules (see Figure 4.7). Despite this cleanup of old toggles, many new toggles were added which explains the overall increasing trend with the total number of toggles doubling between release 22.0 to 34.0.

The number of modules spanned by a feature in release 5 and 13 were higher than the number of features contained in a module. Many features span more than 10 modules in these releases, see Figure 4.7. Release 22.0 and 34.0 saw an increase in both the number of modules and the number of features per module. However, features tended to span fewer modules with the a median of two to five modules per feature. Despite the growth, features tend to be more concentrated in modules.

Below we discuss examples of feature evolution from the module and feature toggles perspectives. For example, in release version 5.0, the base features used to span 6 modules such as `Browser Engine`, `Apps`, `IPC`, `Network`, `SSL` and `NaCl`. However, in release version 22.0 the base feature set was re-factored with toggles moving to the new feature set related to IPC. This change reduced the couple of the base feature set.

A second example, is the touch screen feature that was developed under the toggle *kEnableTouch* and was introduced at Release 6.0. At Release 6.0, the toggle spanned the `UI` and `Browser View` modules. At release 17.0 Chrome developers renamed this toggle to *kEnableTouchEvents*

and moved it inside the `Content` module to implement additional touch based features. Over the next few releases Chrome kept improving touch-screen compatibility by introducing more features with toggles, such as *kEnableTouchpadThreeFingerClick* at release 18.0, *kEnableTouchCallibration* at release 21.0, and *kEnableTouchDragDrop* at Release 25.0. As improvements were made the "Touch Screen" feature now spans many modules, including `Content` and `UI Base`.

A third example is the "Printing" feature, which was developed under the *kPrint* toggle. The feature spanned two modules `UI` and `Printing` before the introduction of `Cloud Printing` in Release 22.0. At release 22.0 *kPrint* toggle was renamed to *kPrintRaster* and it now also spans the `Browser View`.

Examining feature evolution from a module's perspective, the `Content` module illustrates the large increase in feature toggles over time. At release version 13.0, the `Content` module had 113 toggles demonstrating a wide range of content related feature. The features include *kDisableAudio*, *kRendererProcess*, and *kGpuProcess* that are responsible for browser audio disabling, initiating rendering and initiating GPU process, respectively. Over time new features have been introduced to increase the responsibilities of the `Content` module to serve more functionality to the `Render Engine`. For example the *kEnableVideoTrack* toggle was added in release 22.0 to enable video tracking capability. At release 34.0 the number of toggles in `Content` module increased to 285.

Software architecture evolves over time adding new features and functionality. The traditional modular architecture can tell us about this evolution of module dependencies. However, the feature toggle architecture also allows us to understand and explain the evolution of feature dependencies.

4.9 Related Work

4.9.1 Automated Architectural Extraction

Many works have clustered the relationships among files and modules to automatically extract an architecture. Below we discuss some approaches and their limitations.

A comparative study of clustering algorithms for re-modularization of software was conducted by [6]. Clustering architectural components depends upon i) abstract dependencies of entities that

will be clustered, ii) metrics for determining couples between entities and iii) the algorithm for clustering. They found that a proper description scheme is highly important for the entities or modules being clustered. They also propose a novel description scheme as well as better formal evaluation methods for clustering results. Other researchers have also investigated cluster algorithms for architectural extraction, including [91], [5], [129], [82], and [104]. Instead of automatic clustering, we use the knowledge and man-power of multiple students to manually extract related entities.

[59] surveyed six automated techniques to extract and compare architectures.

(1) Algorithm for Comprehension-Driven Clustering (ACDC) [127]

(2) Weighted Combined Algorithm (WCA) [91], and its two measurements WCA-UE and WCA-UENM.

(3) scaLable InforMation Bottleneck [5]

(4) Bunch [90]

(5) Zone-Based Recovery [32, 33]

(6) Architecture Recovery using Concerns (ARC) [60]

They found that each technique produced drastically different architectural representations of the same software. Compared to a manually extracted architectural benchmark, even the best two automated approaches (ARC and ACDC) had low accuracy suggesting that manual recovery may be the best strategy. As a result, we use a semi-automated approach to extract the conceptual, concrete, reference, and feature toggle architectures.

Although automated clustering approaches exist, many works use a semi-automated strategy to extract the architecture of a system. For example, [19] were the first to extract the architecture of Linux Kernel using a semi-automated technique as a combination of manual intervention and tool support. They obtained a concrete architecture by manually extracting the dependencies among the call graph, variable references, and the naming conventions of files and directories followed by the use of tools to construct the final architectural representation. We follow a similar strategy to extract Chrome's modular architecture technique. We then apply feature toggles to the architecture to understand feature relationships among modules.

4.9.2 Architectural Views

Kruchten *et al.* described a 4+1 view of the architecture [84]. The 4+1 view provides a model describing software architecture based on multiple concurrent views such as end-users view, developers view, system-engineer's view and project manager's view which resembles the logical, development, physical and process view. In Krikhaar's doctoral research [83], he examined software architectures from additional viewpoints including Logical View, Module View, Code View, Execution View and Physical View. He considered a number of architectural views such as Logical View, Module View, Code View, Execution View and Physical View. Our study contributes a two new views of the software's feature toggle architecture: the features a module spans and the features that are contained in a module.

4.9.3 Feature Architectures, Product Lines, and AOP

Many works have studied the software feature architectures. However, researchers studied usually study software systems that are developed in feature oriented way or following the product-lines methodologies. In contrast, feature toggles do not require an explicit product line methodology and simply introduce live flags into the code.

Software product lines allow for the release of variations in a software system based on use cases. A product line method FeatureRSEB was developed by Griss [62] combining the two ideas of domain analysis (FODA) [73] and reuse based development. FeatureRSEB performs an exclusive mapping between feature, modules, and architectural elements through traceability links that are related to the use-cases of the software [121]. Use cases ultimately point to the classes inside the system within the architectural elements which allows developers understand the mapping between the system and the various product lines. In contrast, Chrome uses a simple feature toggles approach to allow features to be mapped across multiple modules. Toggles can encapsulate any code feature and are not dependent on specific use cases allowing for greater flexibility.

[37] studied variability in the Linux Kernel. The study focused on information changes in variability models, assets, and mappings between variable features and actual assets. To track down variability they consider the *"KConfig"* files and the pre-processor based `C` code files as the assets

and the "Make" files as the mapping. Compared to their work, we extract feature view of the Chrome architecture considering the toggle configuration files as the feature sets and the toggles inside those files as the representative of features.

Aspect Oriented Programming (AOP) approaches use different techniques to achieve multiple separation of concerns [80],[11]. Traditionally AOP focuses on code level (class files, UML etc.). [71] focus on the cross-cutting concerns of design decisions at the architectural level where features impose design decisions and the cross-cutting concerns. In contrast to Aspects, feature toggles do not require any modifications to the programming language but still allow developers to cross-cut multiple modules and turn on/off certain cross-cutting features. The feature toggle architecture allows us to identify the features within the source code and extract the feature's view of the modular architecture extracted from the source code.

4.9.4 Feature Location and Traceability Recovery

When a developer is working on a newly assigned task he or she needs to understand where to start implementing the new feature. Identifying an initial location for a feature within the source code is commonly known as feature location or concept location [38]. A survey on feature location by [38] found that "feature location" is one of the most important and common activities performed by programmers during software maintenance and evolution. There has been much research into feature location using the dependency graph generated from the source code [24], traceability link recovery between documentation [87, 98], impact analysis [106], and aspect mining [75]. The toggle architecture combined with the modular architecture allows a developer to locate feature code across modules. Looking at the feature in a particular module will inform a developer where to being making modification, *e.g.,* the UI module for a password related feature.

Identifying traceability links between source code and documentation has received considerable attention. **Information retrieval** techniques have been widely used to resolve the links between source code elements and documentation. For example, [7] apply a probabilistic and Vector Space Model (VSM) to resolve terms, while [92] use Latent Semantic Indexing (LSI). Unfortunately these approaches suffer from low precision and recall with LSI having an average precision of 0.42 and

recall of 0.38, and VSM, with an average precision of 0.23 and recall of 0.31 [[8]]. Recent techniques that depend on language specific context have achieved higher precision and recall but are only implemented for Java [36], [113] making them inappropriate for our case study of Chrome.

4.10 Threats to Validity

We studied a single project the Google Chrome web browser, so our results many not generalize to other projects. Since the extraction of the architecture of a large system takes months, the choice of a single project seems reasonable. Our methodology is applicable to any project that uses feature toggles.

On Chrome, feature toggles are not always permanent in the source code. For example, when a new feature is implemented the old feature and all related toggles may be removed. To partially deal with this issue, we extracted the architecture from four different release versions spanning 30 release versions (5.0 to 34.0) in Chrome history. Since toggles are grouped into sets, it is unlikely that the entire feature set will be removed, so we should still have one dependency among the modules which is with the toggles set itself. We may lose some granularity of features because we study four release versions. The more release versions we observe the more accurate information about features and their nature of expanding into modules we explore. However, this need for exploring additional release versions must be balanced against the manual effort.

The mapping of concepts in the documentation to source files and directories was performed as part of a class project by four graduate students and lead by the first author and supervised by the third author. Like any human activity, this process is prone to human error since manual effort is involved. To alleviate the threat due to human error, we had multiple people work on the mapping and resolve inconsistencies through discussion.

4.11 Conclusion

In conclusion, we extracted four architectures for Google Chrome: the conceptual, concrete, browser reference, and feature toggle architectures. The extraction involved examining the conceptual modular entities in the documentation, the call graph relationships, the modules that are common to multiple modern browsers, and the feature toggle relationships among modules. We also examined the evolution of the feature toggles architecture over time. As can be seen in Figure 4.2, the extracted architectures are derived from different but related sources and serve to triangulate each other as the concluding discussion below shows.

Google Chrome did not have an existing conceptual diagram describing the relationships between system entities, so we derived one by examining and discussing the documentation and online forums. Figure 4.3 shows the 19 conceptual modules and 21 edges we identified, including the Browser View, Data Persistence, Browser Engine, and Render Engine modules. This conceptual diagram shows the entire architecture of Chrome and indicates a clear cohesiveness and separation of modules and relationships.

The concrete architecture describes the call graph relationships of the source code between modules. We derive it by manually mapping each source file and directory to those found in the conceptual architecture. This manual mapping took 4 person-weeks. We then extracted the call graph from the source files to determine relationships among modules. In the resulting architectural diagram in Figure 4.4 we identified 5 modules not discussed in the Chrome documentation and 24 modules in total with 72 call edges. We noted some interesting source code violations of the conceptual architecture. Some violations bypassed interfaces to make the system more efficient, while others were related to legacy code, such as the development work to replace the aging Netscape flash plugin.

The browser reference architecture describes the modules required in a modern web browser. We derived this architecture from the conceptual, concrete, and 12 year old [63] reference architecture. In Figure 4.5 we see that while many of the major modules remain the same, modern browsers have a sophisticated plugin module to allow external developers to create add-ons. Furthermore, technological changes, such as GPUs, have introduced new modules.

The feature toggle architecture is extracted from source code and shows which features affect which modules and vice versa. Feature toggles are named "switches" that cover the code that implements a feature. They are used by large web companies to quickly enable and disable a feature in production [109]. Using the directory to conceptual module mapping derived for the concrete architecture, we extract the toggles contained in each module. Toggles that are contained in multiple modules indicate a feature relationship between these modules. In Figure 4.6 we see that there are 1,173 feature relationships between modules. These feature dependencies do not necessarily indicate modular architectural violations as features must span multiple modules. For example, the "EnableAutoFill" feature toggle must be present in the UI module to allow the user to select the content to autofill and also be present in Data Persistence module to store potential autofill content. Our novel contribution of extracting the feature toggle architecture allows researchers and developer to view the features that are present in a module and the modules that are necessary for a feature.

This work has many audiences. Those interested in teaching or working on web browser technology now have a reference to the major architectural modules as well as an case study of one of the most popular web browsers. Developers and researchers who are working on or studying systems with feature toggles now have a method and technique to view the impact of toggles on the modular architecture of the system. Finally, this work required substantial manual effort to extract multiple architectures, we hope that future researchers will use our replication package [108] and file to module mappings to test their automated architectural extraction tools and techniques.

Chapter 5

The Impact of Failing, Flaky, and High Failure Tests on the number of Crash Reports associated with Firefox Builds

Note: The work in this chapter will be submitted to the Foundations of Software Engineering (FSE'18) industry track.

Abstract Testing is an integral part of release engineering and continuous integration. In theory, a failed test on a build indicates a problem that should be fixed and the build should not be released. In practice, tests decay and developers often release builds, ignoring failing tests. In this paper, we studying the link between builds with failing tests and the number of crash reports on the Firefox web-browser. We find that 73% of all crash reports are linked to the builds that ignore test failures. In the median case there are only two crash reports for builds with all tests passing. In contrast, we find that builds with one or more failing test are associated with a median of 517 and 234 crashes for Beta and Production builds, respectively. We further investigate the impact "flaky" tests. Flaky tests are tests that can both pass and fail on the same build. We find that builds with failing flaky tests have a median of 514 and 234 crash reports for Beta and Production builds. Finally, building on previous research that has shown that tests that have failed frequently in the past will fail frequently in the future, we define *HighFailureTests* as those that have failed in 10% or more of their test runs.

Builds with *HighFailureTests* have a median of 585 and 780 crash reports for Beta and Production builds. Unlike other types of test test failures, *HighFailureTests* have a larger impact on Production releases than on Beta builds. We conclude that ignoring test failures is related to a dramatic, 117 to 390 times, increase in the number of crash reports compared with the crash reports on builds with only passing tests.

5.1 Introduction

Software builds are tested to ensure that the functionality of the system is not broken by a change. Developers write test cases when they are developing new features or fixing bugs. In a rapid release model, fixed and shortened release schedules reduce the time for investigation of the test regressions [97]. We examine the impact of ignored failing tests, flaky tests, and *HighFailureTests* on the build quality as measured by the number of user crash reports associated with a build.

We organize our research around the following questions:

(1) **RQ1, Number of Crashes: How many crashes are there for builds on dev, beta, and production?**

Firefox stages its development into three channels. The development contains the current work being done by developers. The beta channel is used by early testers and users. The production channel is released to end users. The stability of the code and the number of users increase as we move from the Dev to Production channel. This first research question quantifies the number of crashes on each channel. This basic information is important to put the remaining research questions into context as low use channels will likely have few crashes but may not be of high quality.

(2) **RQ2, Test Failures: How many crashes are associated with builds that contain test failures?**

This research question quantifies the impact of test failures on crashes. Our goal is to understand if ignored test failures lead to an increase in end user crash reports.

(3) **RQ3, Flaky Tests: How many crashes are associated with builds that contain flaky tests?**

Flaky tests fail non-deterministically[89]. For example, a test may both pass and fail on the same build. As a result, developers cannot trust a flaky test to determine software quality. Our goal is to understand if ignored flaky test failures lead to an increase in the number of browser crashes.

(4) **RQ4, Historical Failures: Do failures of tests that have failed many times in the past lead to an increase in crashes?**

Researchers have shown the tests that have failed in the past tend to continue to fail at high levels [81]. These *HighFailureTests* allowed researches to re-order tests based their historical likelihood to fail [43]. We consider tests that historically fail 10% of the time to be *HighFailureTests*. We investigate whether failures of these tests lead to an increase in the number of crash reports.

The paper is organized as follows, Section 5.2 provides the background on the Firefox project's build process and crash collection. Section 5.3 describes the research methodology. This section also describes the data used in this case study. Section 5.4 discusses the results for each research question. Section 5.5 positions our work in the literature on build systems and testing. Section 5.6 concludes the paper.

5.2 Firefox Release Process

Firefox is a popular open source modern web browser and has been funded by the Mozilla Corporation since November 2004. Firefox's release process involves three channels: Development, Beta[1], and Production [53, 79, 77]. Code remains on each channel for six weeks before transitioning to the next channel. Each channel has a different level of stability, purpose, and number of developers and users who exercise it. McIntosh *et al.* [94] found that the number of users per channel is 100K for Development, 1M for Beta, and 100M+ for the Release channel.

To create a release, a continuous integration tool, Buildbot, is used through `Bootstrap` automation scripts to build newly committed features into a new release [51]. The Buildbot master

[1]Beta was original divided into two channels: Aurora and Beta

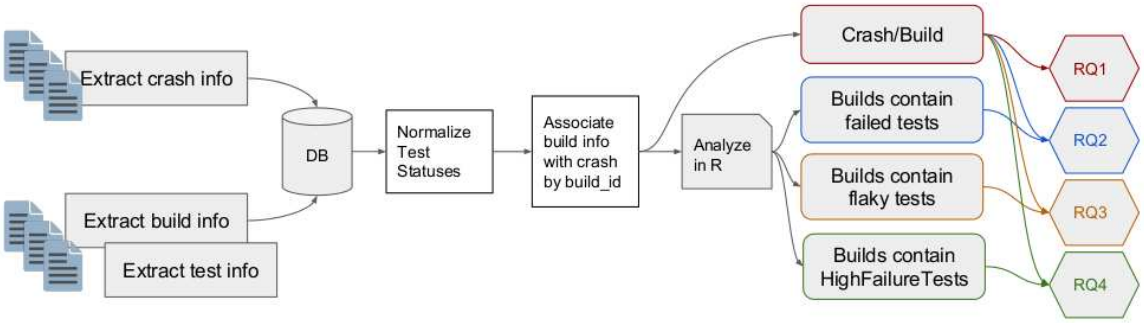


Figure 5.1: Steps in our Research Methodology

creates the build logs and manages the overall process. Each build has a report that contains the logs for each build and includes basic information about the build-setup, environment, test steps, the test verdict, and the overall build result.

When the browser closes unexpectedly a dialogue box allows users to submit crash reports [52]. Each submitted crash report contains a crash dump including the crashing page address, user's local environment, and the Firefox build id.

5.3 Methodology and Data

Our goal is to investigate the impact of ignored failing tests and flaky tests on the number of reported end user browser crashes. We follow a straight-forward method for our study. After loading the data into a database we normalize the test status into three categories: "Pass", "Fail" and "Flaky". We calculate which tests are historically *HighFailureTests*. We then link the build and crash reports based on the build id. We use R to provide statistical answers to our research questions. Figure 5.1 illustrates our research methodology.

5.3.1 Data

We collect the historical build logs and crash reports for Mozilla Firefox spanning from December 2010 to December 2012. We parse the build logs and store the extracted information in a database. The top portion of the log file contains the basic build summary including information about the builder, slave process, start time, pass or fail verdict, build id, and source code revision

number (commit hash).

The test information is contained at the end of the build log file and includes the test status, test path, and a short description of the test. We use the test path to uniquely identify each test. The path is a URL that is linked to the test steps.

We then parse and extract all the crash reports into the database. Each crash report contains a crash signature, URL with an unique id, build id, operating system and other information that may be useful to developers. Once we load the data into the database, we remove incomplete data-rows that have missing information, such as the crashes with no build id.

After extracting the build logs and crash reports we have two data sets containing the crash information and the build history. The attributes are listed in Table 5.1. By joining the two data sets of builds and crashes we extract the builds that could be mapped with one or more crashes. For each build we extract the test steps from the build log and store them separately. We link them based on *build_id* and we found a total of 2.8K unique build ids that have both crash and test information. Associated with these builds are 729K crashes.

Table 5.1: Attributes for the build and crash data

Build Data Attr.	**Crash Data Attr.**
build_id	build_id
build_uid	url
revision	uuid_url
start_time	crash_date
test_info	signature
test_description	-
test_name	-

5.3.2 Test Status Mapping

We found six statuses that Firefox developers use to label their tests. Since there is no formal definition for these test labels, we examined the code of the test scripts [50] [49]. For this paper, we map the test statuses into three categories: Pass, Fail, and Flaky tests. The mapping between Firefox statuses and the categories is found in Table 5.2 along with the number of test runs associated with each status.

Table 5.2: Mapping between Firefox test status and categories used in this paper

Firefox Test Status	Normalized Categories	Number of test verdicts
PASS	*Pass*	120M
PASS(EXPECTED-RANDOM)	*Flaky*	265K
KNOWN-FAIL(EXPECTED-RANDOM)	*Flaky*	10K
KNOWN-FAIL	*Fail*	2M
UNEXPECTED-PASS	*Fail*	1K
UNEXPECTED-FAIL	*Fail*	705

In Table 5.2 the status "PASS" maps to a normal test pass. The "KNOWN-FAIL", "UNEXPECTED-PASS", and "UNEXPECTED-FAIL" are categorized under the "Fail" category. These tests are either problematic long-term failing tests, such as "KNOWN-FAIL" or tests that are expected to pass but actually fail. In contrast, the "PASS(EXPECTED RANDOM)" and "KNOWN-FAIL(EXPECTED RANDOM)" are seen as failing and passing non-deterministically and we consider them to be flaky tests.

5.3.3 Identifying Flaky Tests

Flaky tests non-deterministically lead to a pass or fail verdict. Lou *et al.* identified flaky tests in their study [45] by searching for the key-words "intermittent" and "flak" within the commit history. They used commit logs for identifying flaky tests because they were mostly interested in flaky tests that are already fixed. However, we do not use a keyword search to identify flaky tests. We use the existing Firefox classification in the build log. In the test logs the tests that are marked as "*-RANDOM", we include them in the "Flaky" category which means, tests that are labelled with the statuses "PASS(EXPECTED RANDOM)" and "KNOWN-FAIL(EXPECTED RANDOM)" are considered to be the flaky tests (See table 5.2).

5.3.4 Identifying HighFailureTests

The distribution of failures is not normal. Certain *HighFailureTests* account for a large proportion of total failures. Previous works have used this historical property to re-prioritize tests so that those that have failed frequently in the past will be run first [81, 43]. We investigate if builds with *HighFailureTests* have an increase in the number of crash reports.

We define a *HighFailureTest* to be one that has failed on 10% or more test runs. As an example from the Firefox data set, a test "brokenUTF-16" ran 131K times and 79% of the total run resulted with a "Pass" while 21% times it resulted as a "Fail". We consider this test to be a *HighFailureTests*. In contrast, the test "*hiddenpaging*" which ran 275K times passed 97% of the time with only 3% failures. This test would not be considered a *HighFailureTest* even though it has failed on past builds.

5.4 Results

5.4.1 RQ1: Number of Crashes

How many crashes are there for builds on Dev, Beta, and Production?

A violin plot in Figure 5.2 shows the distribution of crashes for builds in development, beta and production channels. A boxplot showing the median and 25 and 75th quantiles is also shown in the figure.

In total their are 2.8K builds associated with one or more crash reports and 3.8K builds that do not have any crash reports. Although the Dev channel contains experimental code and is likely not as stable as the other channels, we see fewer crashes on this channel. In the median case, development builds are associated with 0 crashes with 3 crashes at the 75th percentile. The Beta channel builds are associated with a median of 437 crashes, and the Production builds are associated to 233 crashes.

Since builds on the Development channel are not typically run by main-stream end users, the number of total users is less likely explaining the limited number of crash reports for builds on this channel. As a result, we do not consider development channel in the remainder of this paper.

The purpose of the Beta channel is to stabilize code. Early adopters use these builds and provide crash and bug reports to help developers to stabilize the code. Despite having fewer users than the Production channel[94], builds on this channel have the highest number of crashes.

Code that reaches the Production channel has passed through various stabilization and bug fixing stages which are intended to reduce the number end user crashes. Although there are many crash reports, given the expanded number of users the production code does appear to be the most stable.

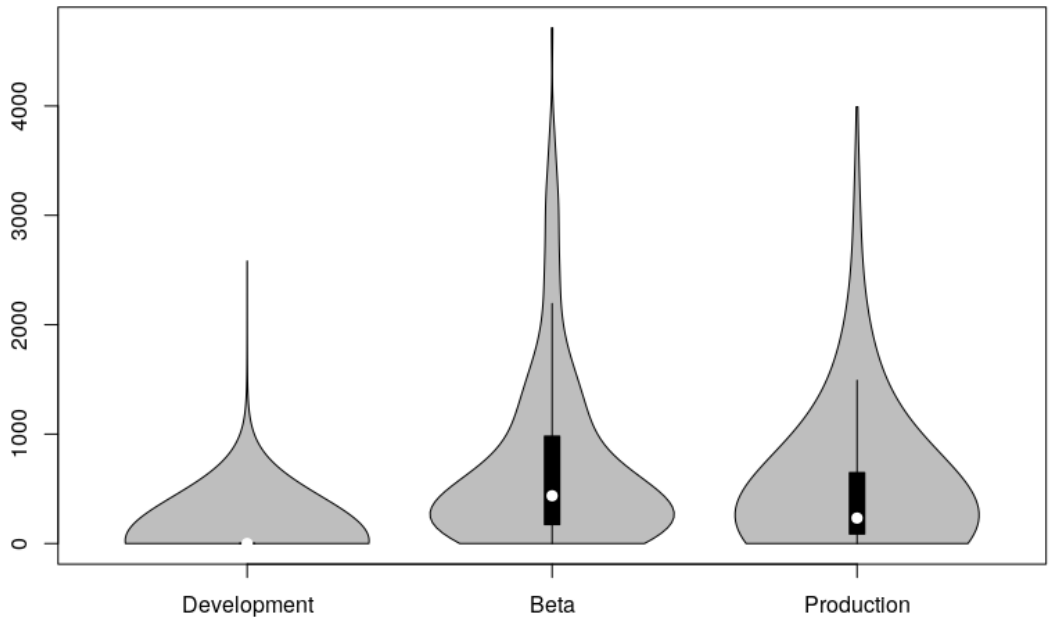


Figure 5.2: Number of crashes for each channel

There are a median of 437 and 233 crashes for builds on the Beta and Production channels. Despite having more end users on the Production channel, there are fewer crashes likely indicating that production code has high stability.

5.4.2 RQ2: Test Failures

How many crashes are associated with builds that contain test failures?

Our conjecture is that when developer ignore quality assurance indicators there will be more crashes on these builds. In this research question, we examine the number of ignored test failures for builds and relate them with the number of crashes. We expect that more browser crashes will be associated with builds that have failing tests (*i.e.* failing builds) compared to those that do not have failing tests (*i.e.* passing or clean builds).

Firefox runs a large number of tests during each build. In the median case 3M tests are run on

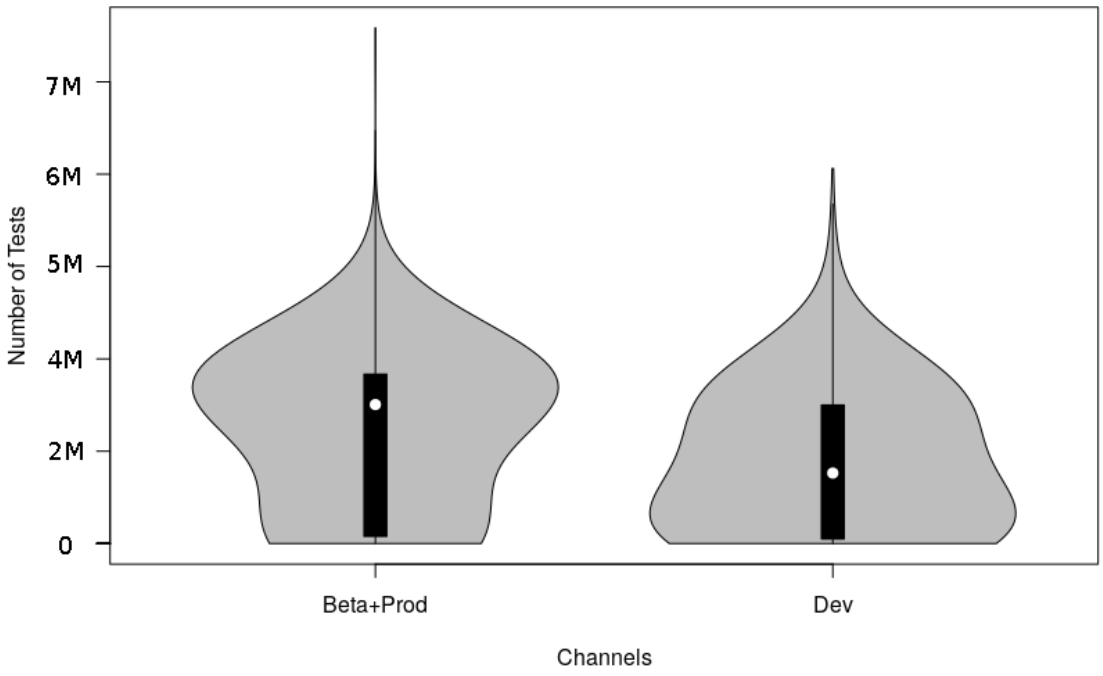


Figure 5.3: Number of tests per build

each build with a maximum of 11M as shown in Figure 5.3. Table 5.2 shows the number of passing and failing tests: 120M and slightly over 2M, respectively. We exclude "random", non-deterministic tests examining them in the next section.

Figure 5.4 contrasts the number of crash reports for builds with at least one failing test with builds that have only passing tests. For the Beta channel, we found that builds that have failing tests have a median of 447 crash reports. In contrast, passing test builds have a median of 2 crash reports with 6 at the 75th percentile and a maximum of 30. In median case failing test build sees 223 times as many crash reports than the median passing test build.

In the Production channel build with failing tests have a median of 247 and the passing build associates 2 crashes in the median case. Failing test builds have 123 times more crash reports than the passing test builds on Production.

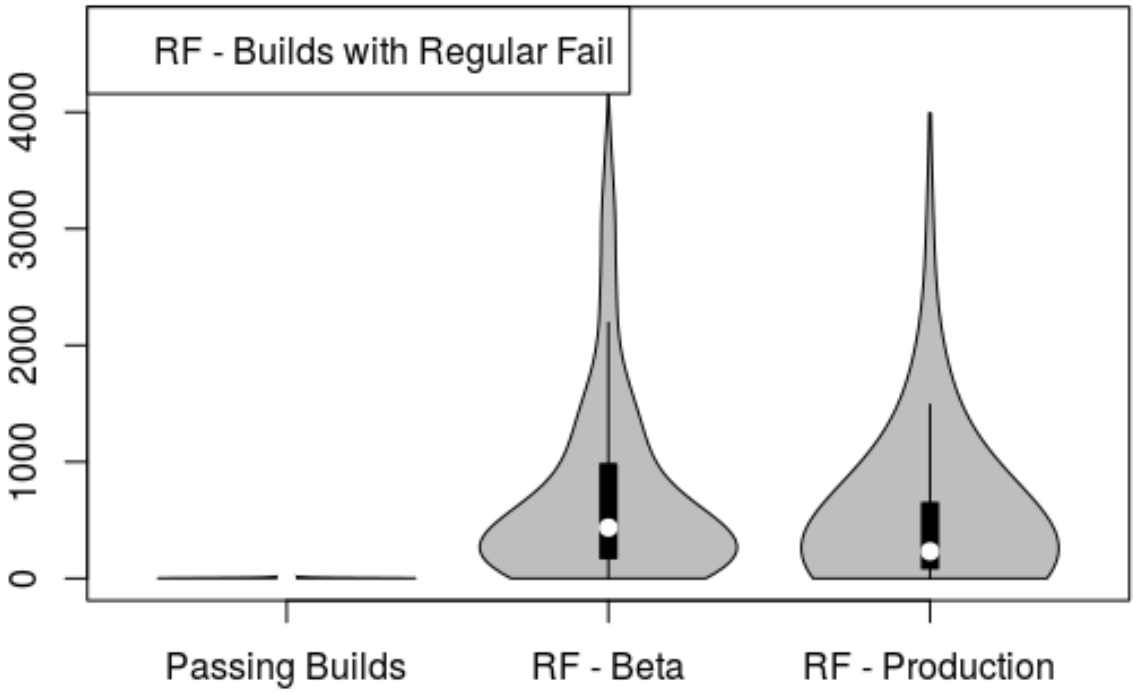


Figure 5.4: Failing Test Builds vs Passing Test Builds for the Beta and Production Channels

In the median case, builds that have failing tests are associated with 447 and 247 crash reports the Beta and Production channels. In contrast, builds with passing tests have a median of only two crash reports.

5.4.3 RQ3: Flaky tests

How many crashes are associated with builds that contain flaky tests?

Flaky tests fail in a non-deterministic manner and potentially hide bugs. For example, if a flaky test fails frequently, developers tend to ignore the failures and could miss the real bugs. We investigate whether ignoring flaky tests is a potential reason for increased browser crashes. We use the Firefox test outcome labels that contain "-RANDOM" to determine which tests are flaky (See the Methodology Section 5.3 for more details on the process of classification). In table 5.2, we see that 275K flaky test-runs are labelled with the "-RANDOM" verdict across all builds.

In the median case each Production channel build that contains at least one flaky test failure is associated with 234 crashes. There is a high degree of variation with 585 crashes at the 75th

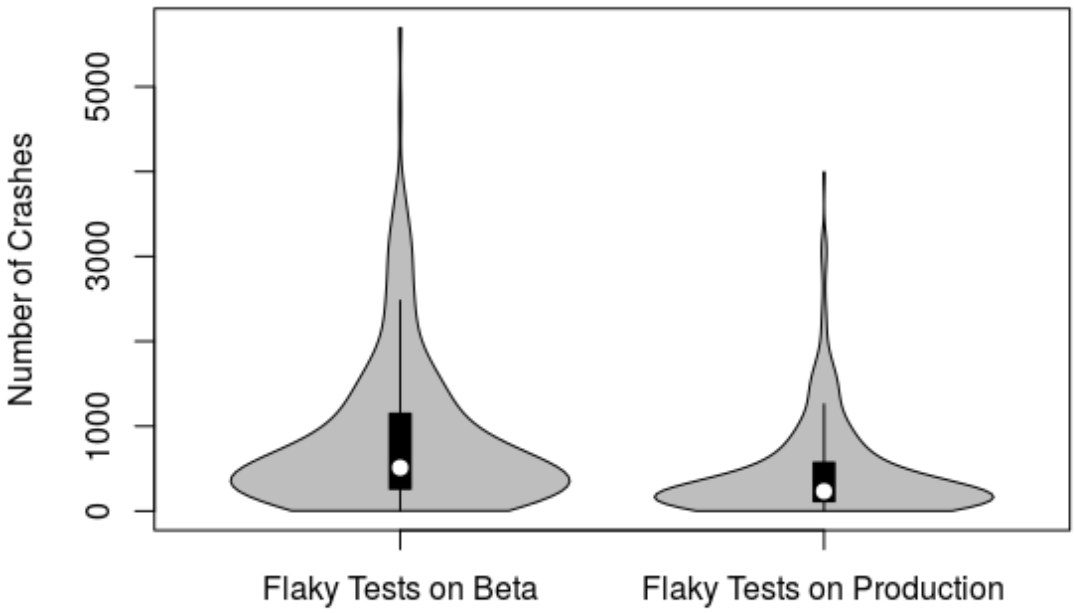


Figure 5.5: Number of crashes for builds with flaky tests

percentile and a maximum of 93k crash reports. Production builds with flaky tests are associated with almost the same number of crashes (247 or 1.05 times) compared to the builds with regular failing tests. However, builds with flaky tests are associated with 117 times more crashes than the passing builds. Figure 5.5 shows the crashes for the builds that contain flaky tests.

For the Beta channel we observed a similar patter with a 514, 1.2K, 21K crash reports for the median, 75th percentile and maximum, respectively. Beta builds with Flaky tests are associated with 1.2 times more crashes than builds with regular failing tests and 257 times higher than the passing builds.

A Wilcoxon test comparing the crashes for flaky builds shows a statistically significant difference between the Beta and Production channels with the p-value $p < 2.6e^{-16}$. While future work is necessary, we conjecture that developers are more conservative with releasing production builds with flaky tests than with beta builds.

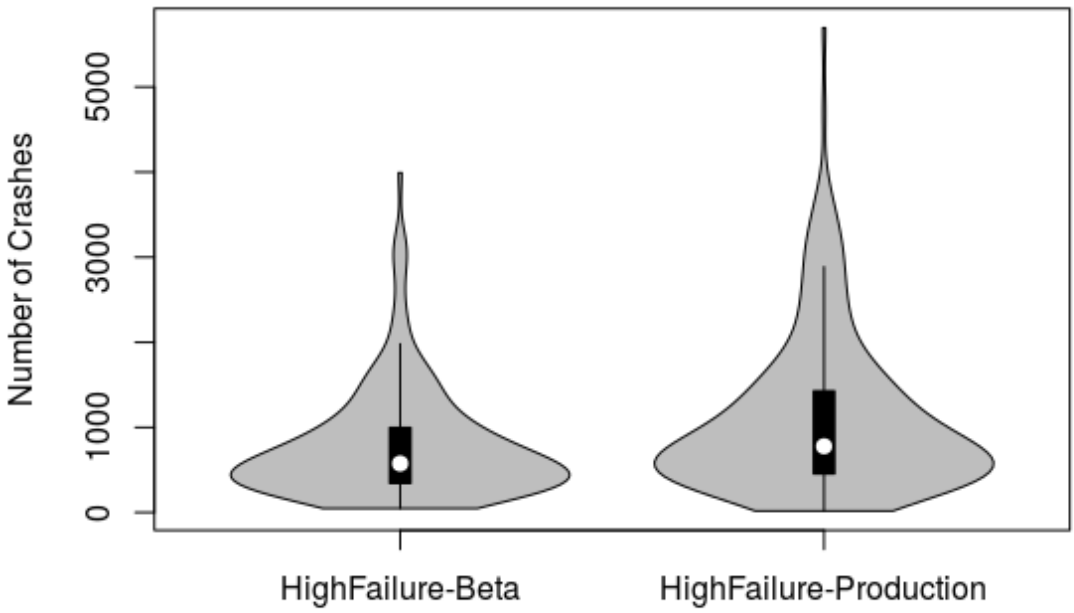


Figure 5.6: Number of crashes for builds with historically *HighFailureTests*

> *In the median case, builds with failing flaky tests 514 and 234 crash reports for Beta and Production, respectively. Compared with normal failing test this represents 1.05 and 1.2 times more crashes.*

5.4.4 RQ4: Historically HighFailureTests

Do failures of tests that have failed many times in the past lead to an increase in crashes?

In software system problems cluster around defective code and tests that have failed frequently in the past are likely to fail in the future [81, 43]. To investigate these tests, we classify test that fail in 10% or more of their total runs as historically *HighFailureTests*. In Figure 5.6, we show that builds that have a failing test that is classified as *HighFailureTest* lead to lower quality builds as measured by an increase in reported crashes on both the Production and Beta channels.

The crash distribution for production builds that have one or more failing tests that are categorized as *HighFailureTests* show a medium, 75th percentile, and maximum of 780, 1.4k, and 21k crash reports, respectively. Production builds with *HighFailureTests* are associated with 3 times more crashes than builds with regular failing tests and 390 times higher than the passing builds.

For Beta builds the corresponding values are 585, 1.5k, and 92k crashes reports for the median, 75th percentile, and maximum, respectively. Beta channel builds *HighFailureTests* are associated with 1.3 times more crashes than builds with regular failing tests and 292 times higher than the passing builds.

A Wilcoxon test comparing *HighFailureTests* on the Beta and Production channels shows a statistically significant difference between the crashes on these two channels with the p-value $p < 0.022$.

Unlike the flaky tests that are labelled by Firefox developers, *HighFailureTests* are not differentiated from other types of failing tests by the developers. Since the number of crashes associated with *HighFailureTests* the highest of the conditions we study, Firefox developers would benefit from identifying and monitoring this classification of tests.

> *In the median case, builds that contain failing historically* HighFailureTests *are associated with 1.3 and 3 times more crashes than the builds with regular failing tests for Production and Beta respectively. Compared to passing builds they are associated with 292 and 390 times more crashes, respectively. There are more Production crashes than Beta crashes for builds with* HighFailureTests.

5.5 Related Work

We divide the related work into testing and build maintenance and quality. We are unaware of any work that has studied the impact of testing on field crash reports.

Testing and Flaky Tests Labuschagne *et al.* studied the cost of regression testing in practice [85]. They found that 18% of the total test suite executions fail. More interestingly, 13% of these failures

are flaky. Of the non-flaky failures, only 74% were caused by a bug in the system under test and the remaining 26% were due to incorrect or obsolete tests. They also found that in the failed builds, only 0.38% of the test case executions failed and 64% of failed builds containing more than one failed test. This study illustrates the importance of dealing with the flaky tests to improve the quality of the regression testing. Our study adds to this knowledge by studying the impact of field crashes instead grouping categories of test failures.

Recent works on flaky tests identified the root cause of the flakiness. For example, Eloussi identified flaky tests from test results [45] in her doctoral research where she proposes three improvements for the basic technique to identify flakiness of tests. By manually examining the flaky test Eloussi divided the tests into three types: Non-Burstly, Burstly and State-Dependent Burstly. Another study on flaky test by Memon *et al.* [95] provides a detail post-mortem of flaky tests that provides actionable information about avoiding, detecting and fixing these types of non-deterministic tests. They inspect the test code by analyzing the code commits that likely fix flaky tests. They also identified the root causes of flakiness but not the impact after release. In our work, we measured the impact of flaky tests on crashes. Future work could use the crash reports on flaky tests to validate the causes identified by Eloussi and Memon *et al.*.

General build system studies Xin *et al.* performed an empirical study on bugs in build systems [130]. They categorized bugs based on their type and severities and found that the third highest percentage of bugs belong to build-configuration category. They examined the association between the bugs and the build configurations, while we associate browser crashes with the failing tests in a build.

McIntosh empirically studied build systems in his dissertation [93]. His publications include a build maintenance study of the effort spent on maintaining the build process and the ownership of these build scripts [94]. He found that maintaining the build system required significant effort, with an overhead of 27% on source code development and 44% on test development. Our work adds evidence that build maintenance is an important problem by showing that ignoring build problems leads to substantial more crashes increasing developer effort and impact on end users.

5.6 Conclusion and Future Work

In this paper, we investigate the association between builds and browser crashes on Beta and the Production channels of the Firefox web browser. We study the impact of ignoring failing, flaky, and *HighFailureTests* on the number of crashes for a build.

We observe that ignoring failing tests makes the Firefox builds much more crash prone compared to the builds that do not have any failing tests (passing builds). Passing builds have a median of 2 crashes for both Beta and Production. In contrast, builds with failing tests have 447 and 247 crashes in the median case, respectively. Flaky tests non-deterministically pass or fail reducing developer confidence in the test. In the median case, builds with failing flaky tests had 514 and 234 crashes for the Beta and Production channels, respectively. Previous works have shown that tests that have failed in the past are more likely to fail in the future [81, 43]. We quantified *HighFailureTests* as those that have failed in 10% of past runs. In the median case, builds with failing *HighFailureTests* are have 585 and 780 crashes for Beta and Production.

Our results show that ignoring failing and flaky tests results in more crashes in Beta than Production. However, ignoring *HighFailureTests* tests leads to more crashes on the Production than Beta channel. Ignored *HighFailureTests* were associated with a median of 780 crashes on the Production channel. This is the most crashes associated with any type of failing test and channel. A failing *HighFailureTests* clearly warrants a detailed investigation before release by Firefox developers.

We hope that our work will inspire developers to understand the high risk of ignoring failing tests. We hope that researchers will extend our work by examining both the root causes of failing and flaky tests and by contributing advanced statistical models to enhance our understanding of the associated risks.

Chapter 6

Future Works

In this dissertation, we have presented the results from the empirical investigation on modern release engineering practices. Adopting rapid release and single trunk based development is accepting large popularity by the practitioners. Single trunk based development helps to achieve the Continuous Integration process where feature toggle is a powerful technique to postpone the availability of a feature from the trunk. Many practitioners are talking about feature toggles and their various usage. Our study uncovers a number of information and statistics that validate most of the claims by the practitioners who used feature toggles in their daily work. We also studied the impact of using feature toggles on software architecture where at this point we performed this study on only one large software system. We performed a case study on the impact of ignoring failing and flaky tests which is at a preliminary level. These investigatory case studies on different areas of release engineering opens the door to much in-depth research directions. In this chapter, we discuss the potential areas where further study may be required to uncover more useful information in the context of release engineering.

6.1 Comparing the Effort Involved in Using Feature Toggles and Feature Branches

It frequently asked question regarding the use of feature toggles is related to the effort that is involved in using feature toggles compared to using feature branches. An empirical study would

answer this question by investigating software projects preferably the proprietor industrial software systems. Both feature branching and feature toggle strategies have proper reasons to choose one over another. For example, one may not like to use feature toggles because it takes a good amount of effort to keep the toggles organized and also there are risks of using feature toggles. Moreover it creates dead code which demands huge effort to keep the code-base clean by maintaining the toggles on a regular basis. Compared to that, "merge hell" in branching system is a nightmare to many release engineers as they try to combine outdated branches into a coherent release.

Many companies have already switched to single-trunk based development practice using feature toggles when they found it is difficult to stick to the release deadline while merge conflicts drag the pace down. Therefore, this is still an open debate. Yet, the very common question "*Should I use feature toggles over branches?*" does not have a straight answer. We feel an empirical study is necessary at this point to identify the factors that will be helpful for the practitioners to take the decision about using feature toggles or feature branches.

6.2 Case Study on Feature Toggles in Web Applications

In Chapter 3 we performed a case study on Google Chrome web browser which is an open source desktop-based software that allows us to browse the internet. A similar case study on web applications would be great while feature toggles are used in web applications robustly [3].

Web applications differ substantially from mobile applications, not only in terms of the system design (business logic) but also in terms of the infra-structure and deployment organization [57]. Therefore, the usage, scope, and robustness of feature toggles might be different in these two different types of software systems. Feature toggles are often used in canary deployment [10] to safely roll-out new web based features. For example, by turning a feature toggles off release engineers can roll back to a release version even after the database schema was modified for the new release. Furthermore, most of the web applications provide web services, a single applications may provide multiple services, at the same time, a single service may have multiple groups of customers. DevOps engineers use feature toggles for their different conditional configurations to control the flow of the services of their applications towards different gorups of clients [70]. Therefore, feature toggles are

more popular in web applications [3] and a case study bring insight into this feature management technique.

6.3 Dynamic Analysis to Uncover Toggle Coverage

In our work, we used static analysis to determine feature toggle coverage. Chrome is written in C++ which makes coverage analysis difficult due to pointers. Dynamic analysis have been proven its successful use in the field of software reverse engineering and code analysis for better understanding [122, 103]. Applying dynamic analysis techniques to understand the code areas covered by the toggles would likely lead to more accurate coverage measurement.

6.4 Re-Factoring Feature Toggles

Feature toggles allows dead code to remain in a system if developers do not routinely remove old features. Future research on re-factoring toggles is an exciting area. Many researchers in the literature have studied software re-factoring based on popular re-factoring methods such as move method [56], type-checking [125], extract method [126]. Toggles that are supposed to get cleaned up but still persist within the code-base may be introduced as a new type of code smell that we may call "toggle smell". We can categorise the toggles smells into different types based on their characteristics and potential causes and consequences for each of them. The following sub-sections highlight how future work could identify certain toggles smells.

6.4.1 Dead Toggles

This smell could be identified if the toggle repository does not possess the information about the feature toggle, but the source code is still using the toggle to encapsulate code blocks. For example the toggle "*kEnableTextInputFocusManager*" in Google Chrome ("*/ash/root_window_controller_unittest.cc*") is used in an if condition that executes two different statements based on the toggle status.

The feature for which the toggle was added may became obsolete in future and the developers may not remove the toggle associated with the feature from the source code (if systematic toggle-cleanup or maintenance is not performed in a regular basis). Developer may remove the toggle

from the toggle repository unknowingly causing the dead toggle smell to appear. If there are no adequate test cases (functional/unit) to cover the code encapsulated under the toggle flag. If unit test cases were present to cover all the toggles in the toggle repository, this smell could be potentially identified by a test failure.

6.4.2 Nested Toggles

This toggle smell can be observed when one feature toggle is dependent on the state of one or more other feature toggle. For example if the main menu item is not enabled or visible, then the sub menus should not be visible or enabled. So in this scenario the toggle that turns the sub menus on, must be dependent to the toggle that controls the main menu. In this case we may call the toggle for main menu as "Master Toggle". Often, a master toggle is used to turn on/off many slave toggles of the related/dependent features. In some software segment like e-commerce/online gambling/gaming, a set of features will be targeted for audience of a particular region, in these cases a master toggle is used to control the dark-launch for all or some of the features.

6.4.3 Spaghetti (Combinatorial) Toggles

We can observe this toggle smell when the execution of code is controlled by combination of states of several toggles.

The following consequences we can observe with presence of this toggle smell in the project. i). Cost of testing will increase for the code areas covered under such toggles. ii). Changing existing code base will become non-trivial. iii). Code complexity becomes higher and understanding code becomes difficult.

6.4.4 Mix of Compile-Time and Runtime Toggles

This smell is observed when the source code uses multiple techniques to manage toggles. One such usage is to combine compile time (#define, #ifndef) along with the run time toggles.

This is generally observed when the code having conditional compilation prevents unnecessary changes from being included into the target binary (to reduce size). But unfortunately within those

statements the feature toggle statements are also included. As a consequence the cost of testing may increase for all the code paths with different binary versions.

6.4.5 Spread Toggles

We can observed this toggle smell when the same toggle flag is used across several modules or packages as well as across several source files.

This toggle smell may appear if features are not isolated. In case an agile methodology is followed using user stories, the user stories are not decomposed to the possible extent. Another reason can be, if feature becomes a cross cutting concern in the source code or modules. This also indicates that the component architecture of the target system is highly coupled and not easily decomposable. However, this smell is not because of the usage of the toggle methodology. Another possible cause that could lead to this smell is use of master toggle to control the set of relevant features. So in this scenario this smell may become a false positive from the project's perspective.

6.5 Investigate Test Cases to Understand the Reason Behind Test Failure

A test case defines a path of the control flow of the execution of a particular operation for a specific input or input set from the beginning to an end point. Test cases provide a very specific information about a particular case of input for a feature. A collection of test cases eventually represent the control-flow graph of an execution flow for all possible input cases depending on the corresponding business logic that it implements. Therefore, analyzing the test cases would be useful to identify the reason for test failure.

The empirical study in Chapter 5 validates that there is a relation between failing builds and the end user crashes. However, this study is still at a primary investigatory level and do no specify the reason why and how the failing builds relate to crashes. A future study can be performed to investigate and specify the reasons behind the test failures.

6.6 Analyze Crash Reports to Identify Crash Prone Areas

Many software projects have their standard crash reporting system that generates software crash reports in a formated way so that developers can use them if needed. Crash reports contain a number of information that provide almost a complete understanding about the crash event. Our study in Chapter 5 finds the association between browser crash and failing builds. However, the large set of information that we get from the crash reports is a potential source of further investigation on identifying fault prone areas of the software system. The "crash-signature" is the main identifer for a crash report. It provides a "call-stack" which contains potential information about the methods involved in the process. Crash signatures classifies crash reports and put them into clusters. This may help us identifying vulnerable or error prone clusters of the system as well as the tests that need more attentation by the developers.

Chapter 7

Conclusion

Modern release engineering practices have become more rapid and incremental than the early days of software development. In this thesis, we empirically investigate modern release engineering practices. Our investigation covers the developers activity, time and effort spent in different periods of a release cycle. Our study also covers the feature management where we look into feature toggles and how they are used to control the execution of features at a run time of the software system. We also investigate the impact of using feature toggles on the architecture of the software system that allows us to provide a new representation of the software architecture as a new addition to the software architectural view points. Finally, we studied the quality of builds as an impact of ignoring failing and flaky tests during the build.

7.1 Contributions and Findings

We summarize our contributions from each milestone. Our quantitative results from the study on release cycle and developers activity gives us an overall idea about i. the length of release cycle, ii. how the developers react to the release deadline when it approaches closer, iii. how quickly the code-changes for feature development are merged into the main branch compared to the bug fixes. We also measure the effort spent by the developers and ownership of the code in stabilization period.

We studied the feature management in the second milestone of this thesis. We found that leading software companies such as Google developing their Chrome browser in a rapid release model and

managing features using feature toggles. We conduct a mixed-method study on feature toggles to quantitatively and qualitatively understand the usage, life-span, advantage and benefits of using feature toggles.

In the third milestone, we find that feature toggles impact the architecture of the system by controlling the flow of the features. In this study This we extracting a completely new representation of software architecture from features perspective. The feature toggle based architecture uncovers hidden relationships and dependencies among the modules and features. It also shows us the cross-cutting nature of the features and the modules. Throughout the extraction process we extract the conceptual and concrete modular architecture of the Chrome browser based on a traditional approach. We also update the 12 years old browser reference architecture based on the technological improvements and changes in modern web browsing technique. While many of the major modules from the old reference architecture remain the same, modern browsers have a sophisticated plugin module to allow external developers to create add-ons. Furthermore, technological changes, such as GPUs, have introduced new modules in the updated reference architecture.

In the fourth milestone, we quantify the impact of ignore failing and flaky tests on software builds. We found that ignoring failing and flaky tests increases the association between builds and end user crashes more than several hundred times compared to the builds in which all tests pass.

7.1.1 Release Cycle and Developers Activity

Rapid release is a modern practice of releasing new features quickly to the end users. The first part of this thesis investigates the release cycles in Chrome and Linux. We discuss and quantify the effort and time spent by developers and release managers during development and stabilizing. Our finding in this research are listed below:

(1) **Release cycle length varies**: Although Linux and Chrome use regular rapid release schedules, release stabilization lengths of them are almost similar which varies by approximately 10 days. Since they practice different strategies for development, the lengths of the development periods are very different. Although at the initial stage of adopting rapid release schedule Linux and Chrome had a little inconsistent release cycles. However, over time, both projects

have become better at releasing on schedule.

(2) **Effort spent by core developers**: A very small group of developers control the stabilization of a release. Only 23 developers in Linux Kernel and 10 developers for Google Chrome take the responsibility of stabilizing the code changes made by hundreds of developers.

(3) **Integration speed**: Changes made during stabilization period (*e.g.,* fixes to regressions) are integrated and released more quickly than the changes made during the development period.

(4) **Rush to release**: Although in rapid release developers do not have to wait uncertainly for the next up coming release, we found a significant amount of rush in daily commits and churns. Two weeks before stabilization begins, the daily churn rate increases by 20% and 21% for Linux and Chrome respectively.

7.1.2 Feature Management Using Feature Toggles

Once we have investigated the release cycle and developers activity, we perform a case study to understand how features are managed while developers are following a regular rapid release process and developing on single trunk. Since google Chrome is using feature toggles to manage and control their features, we perform a detail and in depth study on feature toggles and quantify the "Adoption", "Usage", "Development Stage" and "Lifetime" of feature toggles. The study on feature toggles gives us the following findings.

(1) **Adoption – How many toggles exist?**: The number of toggles increases linearly over time, except for a period of active maintenance at release 35.

(2) **Usage – How are toggles used?**: Most toggles are used directly in a conditional, however, at least 30% of the toggles are assigned to a variable to allow them to significantly change the behaviour of Chrome, which complicates the maintenance.

(3) **Development Stage – Are toggles modified during development or release stabilization?**: Most of the toggle changes occur during development, only 3% of all toggle changes happen during release stabilization.

(4) **Lifetime – How Long do Toggles Remain in the System?**: In general, the lifetime for toggles is long. More than half of the toggles are surviving 12 or more release version within the source-code of Chrome.

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

7.1.3 Architecture from Features Perspective

The extracted architecture based on the feature toggles carry demostrates the cross-cutting nature of features in Chrome. To provide comparisons with the feature toggle architecture, we extracted the conceptual and the concrete modular architecture of Google Chrome. We update Grosskurth's twelve years old browser reference architecture. The extraction of architecture based on feature toggles is a new addition to the software re-engineering and architecture extraction techniques. The representation of the architecture from the features perspective will facilitate the developers while they will be aware of more concrete and up-to-date state of the features and their relationshiops to other modules in the source code.

7.1.4 The Impact of Ignored Failing and Flaky Tests on the Quality of Builds

In this milestone, we focus our investigation on the association of software crash reports with the builds that contain failing and flaky tests. We performed this study on Mozilla Firefox and we found that software builds that ignores failing and flaky tests, have higher number of association with crashes on the end-user's system. Builds that have failing tests are associated with 123 and 223 times more crash reports than the passing test builds for the Beta and Production channels, respectively.

Builds with labelled flaky tests are associated with 1.2 and 1.05 times more crashes than the builds with regular failing tests, for Beta and Production respectively. Compared to passing builds they are associated with 257 and 117 times more crashes for the Beta and Production channels respectively.

Builds with *HighFailureTests* are associated with 1.3 and 3 times more crashes than the builds with regular failing tests. Compared to passing builds they associate with 292 and 390 times more crashes for the Beta and Production channels respectively.

This thesis focuses on four aspects of modern release engineering practices from the perspective of time and effort, feature management, feature archtitecture and release quality. Since modern software companies are moving towards faster releases and quick delivery to the market, this thesis has been conducted to understand how developers are achieving rapid releases. We hope that both researchers and practitioners will find this work useful in helping them understand and implement rapid releases.

Bibliography

[1] A. MOCKUS, R.T. FIELDING, J. H. A case study of open source software development: the apache server. In *Proceedings of the 2000 International Conference on Software Engineering, 2000.* (June 2000), pp. 263–272.

[2] ADAMS, B., BIRD, C., BELLOMO, S., KHOMH, F., AND MOIR, K. International workshop on release engineering (releng). http://releng.polymtl.ca.

[3] ADAMS, B., AND MCINTOSH, S. Modern release engineering in a nutshell–why researchers should care. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on* (2016), vol. 5, IEEE, pp. 78–90.

[4] ALLSPAW, O. M., AND HAMMOND, P. 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.

[5] ANDRITSOS, P., AND TZERPOS, V. Information-theoretic software clustering. *IEEE Trans. Softw. Eng. 31*, 2 (Feb. 2005), 150–165.

[6] ANQUETIL, N., AND LETHBRIDGE, T. C. Comparative study of clustering algorithms and abstract representations for software remodularisation. *IEE Proceedings - Software 150*, 3 (June 2003), 185–201.

[7] ANTONIOL, G., CANFORA, G., CASAZZA, G., DE LUCIA, A., AND MERLO, E. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering 28*, 10 (2002), 970–983.

[8] BACCHELLI, A., LANZA, M., AND ROBBES, R. Linking e-mails and source code artifacts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (2010), pp. 375–384.

[9] BARR, E. T., BIRD, C., RIGBY, P. C., HINDLE, A., GERMAN, D. M., AND DEVANBU, P. Cohesive and isolated development with branches. In *Fundamental Approaches to Software Engineering*. Springer, 2012, pp. 316–331.

[10] BASS, L., WEBER, I., AND ZHU, L. *DevOps: A Software Architect's Perspective*, 1st ed. Addison-Wesley Professional, 2015.

[11] BATORY, D., LIU, J., AND SARVELA, J. N. Refinements and multi-dimensional separation of concerns. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2003), ESEC/FSE-11, ACM, pp. 48–57.

[12] BENAVIDES, D., TRINIDAD, P., AND RUIZ-CORTÉS, A. Automated reasoning on feature models. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering* (Berlin, Heidelberg, 2005), CAiSE'05, Springer-Verlag, pp. 491–503.

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

[14] BERGER, T., PFEIFFER, R.-H., TARTLER, R., DIENST, S., CZARNECKI, K., WKASOWSKI, A., AND SHE, S. Variability mechanisms in software ecosystems. *Information and Software Technology 56*, 11 (2014), 1520–1535.

[15] BIRD, C., RIGBY, P. C., BARR, E. T., HAMILTON, D. J., GERMAN, D. M., AND DEVANBU, P. 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.

[16] BIRD, C., AND ZIMMERMANN, T. Assessing the value of branches with what-if analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2012), FSE '12, ACM, pp. 45:1–45:11.

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

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

[19] BOWMAN, I. T., HOLT, R. C., AND BREWSTER, N. V. Linux as a case study: Its extracted software architecture. In *Proceedings of the 21st International Conference on Software Engineering* (New York, NY, USA, 1999), ICSE '99, ACM, pp. 555–563.

[20] BRAVO. List of chromium command line switches (for 2012-09-26). http://peter.sh/experiments/chromium-command-line-switches/?date=2012-09-26, 2012.

[21] BRUN, Y., HOLMES, R., ERNST, M. D., AND NOTKIN, D. Early detection of collaboration conflicts and risks. *IEEE Trans. Softw. Eng. 39*, 10 (Oct. 2013), 1358–1375.

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

[23] CATALDO, M., AND HERBSLEB, J. D. Factors leading to integration failures in global feature-oriented development: An empirical analysis. In *Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, 2011), ICSE '11, ACM, pp. 161–170.

[24] CHEN, K., AND RAJLICH, V. Case study of feature location using dependence graph. In *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension* (2000), pp. 241–247.

[25] CHROME. The chromium projects: Design documents. http://www.chromium.org/developers/design-documents, April 2015.

[26] CHROME. Google chrome developers' documentation. https://developer.chrome.com/extensions/devguide, March 2015.

[27] CHROME. The chromium projects: Nacl and pnacl. http://www.chromium.org/native-client/nacl-and-pnacl, February 2016.

[28] CHROME. Content module. http://www.chromium.org/developers/content-module, February 2016.

[29] CHROME. What are extensions. https://developer.chrome.com/extensions, February 2016.

[30] CHROME, G. Saying goodbye to our old friend npapi. http://bit.ly/2fgG8UX, September 2013.

[31] CHROMEWIKI. Google chrome official wiki page. http://en.wikipedia.org/wiki/Chrome, March 2015.

[32] CORAZZA, A., DI MARTINO, S., AND SCANNIELLO, G. A probabilistic based approach towards software system clustering. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering* (Washington, DC, USA, 2010), CSMR '10, IEEE Computer Society, pp. 88–96.

[33] CORAZZA, A., MARTINO, S. D., MAGGIO, V., AND SCANNIELLO, G. Investigating the use of lexical information for software system clustering. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering* (Washington, DC, USA, 2011), CSMR '11, IEEE Computer Society, pp. 35–44.

[34] CSACADEMY. Graph editor. https://csacademy.com/app/graph_editor, 09 2017.

[35] CZARNECKI, K., HELSEN, S., AND EISENECKER, U. *Staged Configuration Using Feature Models*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 266–283.

[36] DAGENAIS, B., AND ROBILLARD, M. P. Recovering traceability links between an API and its learning resources. In *Proceedings of the 34th ACM/IEEE International Conference on Software Engineering* (2012), pp. 47 –57.

[37] DINTZNER, N., V. DEURSEN, A., AND PINZGER, M. Fever: Extracti ng feature-oriented changes from commits. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)* (May 2016), pp. 85–96.

[38] DIT, B., REVELLE, M., GETHERS, M., AND POSHYVANYK, D. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process 25*, 1 (2013), 53–95.

[39] DIXON, E., ENOS, E., AND BRODMERKLE, S. A/b testing of a webpage, July 5 2011. US Patent 7,975,000.

[40] DOCUMENTATION, O. Writing device drivers. http://bit.ly/1LWsLWa.

[41] DUBINSKY, Y., RUBIN, J., BERGER, T., DUSZYNSKI, S., BECKER, M., AND CZARNECKI, K. An exploratory study of cloning in industrial software product lines. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering* (Washington, DC, USA, 2013), CSMR '13, IEEE Computer Society, pp. 25–34.

[42] DUVALL, P., MATYAS, S. M., AND GLOVER, A. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.

[43] ELBAUM, S., ROTHERMEL, G., AND PENIX, J. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), FSE 2014, ACM, pp. 235–245.

[44] ELLIOTT, E. *Programming JavaScript Applications*. O'Reilly Media, June 2014.

[45] ELOUSSI, L. *Determining flaky tests from test failures*. PhD thesis, University of Illinois at Urbana-Champaign, 2015.

[46] ERIK SOWA, R. L. 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.

[47] ETHERTANK, DBARON, KOHEI, AND KENNYKAIYINYU. https://mzl.la/2i7M1vp, April 2013.

[48] FIREFOX. Multi-process firefox - technical overview. https://mzl.la/2AwsuJp, 05 2015.

[49] FIREFOX, M. Test labeling in mozilla : integration-mozilla-inbound. http://bit.ly/2riO0Nh, Jul 2012.

[50] FIREFOX, M. Test labeling in mozilla : Xpcshell self test. http://bit.ly/2scySB1, Jan 2012.

[51] FIREFOX, M. Build:release automation. https://wiki.mozilla.org/Build:Release_Automation, March 2017.

[52] FIREFOX, M. Firefox crash reporter. https://support.mozilla.org/en-US/kb/mozillacrashreporter, March 2017.

[53] FIREFOX, M. Release management/release process. https://mzl.la/1KhlZf9, 06 2017.

[54] FITZ, T. 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.

[55] FITZGERALD, B., AND STOL, K.-J. Continuous software engineering and beyond: Trends and challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering* (New York, NY, USA, 2014), RCoSE 2014, ACM, pp. 1–9.

[56] FOKAEFS, M., TSANTALIS, N., AND CHATZIGEORGIOU, A. Jdeodorant: Identification and removal of feature envy bad smells. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on* (2007), IEEE, pp. 519–520.

[57] FOWLER, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

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

[59] GARCIA, J., IVKOVIC, I., AND MEDVIDOVIC, N. A comparative analysis of software architecture recovery techniques. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (Piscataway, NJ, USA, 2013), ASE'13, IEEE Press, pp. 486–496.

[60] GARCIA, J., POPESCU, D., MATTMANN, C., MEDVIDOVIC, N., AND CAI, Y. Enhancing architectural recovery using concerns. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering* (Washington, DC, USA, 2011), ASE '11, IEEE Computer Society, pp. 552–555.

[61] GLASER, B. G., AND STRAUSS, A. L. *The discovery of grounded theory: Strategies for qualitative research*. Transaction Publishers, 2009.

[62] GRISS, M. L., FAVARO, J., AND ALESSANDRO, M. D. Integrating feature modeling with the rseb. In *Proceedings of the 5th International Conference on Software Reuse* (Washington, DC, USA, 1998), ICSR '98, IEEE Computer Society, pp. 76–.

[63] GROSSKURTH, A., AND GODFREY, M. W. A reference architecture for web browsers. In *Proceedings of the 21st IEEE International Conference on Software Maintenance* (Washington, DC, USA, 2005), ICSM '05, IEEE Computer Society, pp. 661–664.

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

[65] HAMMANT, P. Continuous delivery: A maturity assessment model. http://info.thoughtworks.com/Continuous-Delivery-Maturity-Model.html, March 2013.

[66] HAREL, E. 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.

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

[68] HO, I., AND PASRICHA, L. 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.

[69] HUMBLE, J., AND FARLEY, D. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.

[70] HTTERMANN, M. Building blocks of devops. *DevOps for Developers* (2012), 33–47.

[71] JANSEN, A., AND BOSCH, J. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture* (Washington, DC, USA, 2005), WICSA '05, IEEE Computer Society, pp. 109–120.

[72] JIANG, Y., ADAMS, B., AND GERMAN, D. M. Will my patch make it? and how fast?: case study on the linux kernel. In *Proceedings of Mining Software Repositories* (2013), IEEE Press, pp. 101–110.

[73] KANG, K., COHEN, S., HESS, J., NOVAK, W., AND PETERSON, S. Feature–Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Nov. 1990.

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

[75] KELLENS, A., MENS, K., AND TONELLA, P. A survey of automated code-level aspect mining techniques. *Transactions on aspect-oriented software development IV* (2007), 143–162.

[76] KERZAZI, N., KHOMH, F., AND ADAMS, B. Why do automated builds break? an empirical study. In *Proc. of the 30th IEEE Intl. Conf. on Software Maintenance and Evolution (ICSME)* (2014), pp. 41–50.

[77] KHOMH, F., ADAMS, B., DHALIWAL, T., AND ZOU, Y. Understanding the impact of rapid releases on software quality. *Empirical Software Engineering 20*, 2 (2015), 336–373.

[78] KHOMH, F., DHALIWAL, T., ZOU, Y., AND ADAMS, B. Do faster releases improve software quality? an empirical case study of mozilla firefox. In *Proceedings of Mining Software Repositories* (June 2012), pp. 179–188.

[79] KHOMH, F., DHALIWAL, T., ZOU, Y., AND ADAMS, B. Do faster releases improve software quality?: an empirical case study of mozilla firefox. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories* (2012), IEEE Press, pp. 179–188.

[80] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming* (London, UK, UK, 2001), ECOOP '01, Springer-Verlag, pp. 327–353.

[81] KIM, J.-M., AND PORTER, A. A history-based test prioritization technique for regression testing in resource constrained environments. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on* (2002), IEEE, pp. 119–129.

[82] KOBAYASHI, K., KAMIMURA, M., KATO, K., YANO, K., AND MATSUO, A. Feature-gathering dependency-based software clustering using dedication and modularity. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)* (Washington, DC, USA, 2012), ICSM '12, IEEE Computer Society, pp. 462–471.

[83] KRIKHAAR, R. L. *Software architecture reconstruction*. Philips Electronics, 1999.

[84] KRUCHTEN, P. *Architecture Blueprints&Mdash;the &Ldquo;4+1&Rdquo; View Model of Software Architecture*. TRI-Ada '95. ACM, New York, NY, USA, 1995.

[85] LABUSCHAGNE, A., INOZEMTSEVA, L., AND HOLMES, R. Measuring the cost of regression testing in practice: A study of java projects using continuous integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2017), ESEC/FSE 2017, ACM, pp. 821–830.

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

[87] LINDVALL, M., AND SANDAHL, K. Practical implications of traceability. *Softw. Pract. Exper. 26*, 10 (Oct. 1996), 1161–1180.

[88] LINUX. The linux kernel development process. https://www.kernel.org/doc/Documentation/development-process/2.Process Accessed February 2013.

[89] LUO, HARIRI, E. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), ACM, pp. 643–653.

[90] MANCORIDIS, S., MITCHELL, B. S., CHEN, Y., AND GANSNER, E. R. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the IEEE International Conference on Software Maintenance* (Washington, DC, USA, 1999), ICSM '99, IEEE Computer Society, pp. 50–.

[91] MAQBOOL, O., AND BABRI, H. A. The weighted combined algorithm: A linkage algorithm for software clustering. In *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)* (Washington, DC, USA, 2004), CSMR '04, IEEE Computer Society, pp. 15–.

[92] MARCUS, A., AND MALETIC, J. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th ACM/IEEE International Conference on Software Engineering* (2003), pp. 125–135.

[93] MCINTOSH, S. *Studying the Software Development Overhead of Build Systems*. PhD dissertation, Queen's University, 08 2015.

[94] MCINTOSH, S., ADAMS, B., NGUYEN, T. H. D., KAMEI, Y., AND HASSAN, A. E. An empirical study of build maintenance effort. In *2011 33rd International Conference on Software Engineering (ICSE)* (May 2011), pp. 141–150.

[95] MEMON, C. Automated testing of gui applications: models, tools, and controlling flakiness. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE Press, pp. 1479–1480.

[96] MICCO, J. *Tools for Continuous Integration at Google Scale*. Google Tech Talk, Google Inc., 2012.

[97] MIKA V. MANTYLA, FOUTSE KHOMH, B. A. On rapid releases and software testing. In *Proceedings of the 29th International Conference on Software Maintenance (ICSM)* (Eindhoven, Netherlands, September 2013), IEEE, pp. 20–29.

[98] MIRAKHORLI, M., AND CLELAND-HUANG, J. Transforming trace information in architectural documents into re-usable and effective traceability links. In *Proceedings of the 6th International Workshop on SHAring and Reusing Architectural Knowledge* (New York, NY, USA, 2011), SHARK '11, ACM, pp. 45–52.

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

[100] MOZILLA. Mozilla release engineering. https://wiki.mozilla.org/ReleaseEngineering#Team Accessed December 2014.

[101] MOZILLAWIKI. javascript:spidermonkey:odinmonkey. https://mzl.la/2A4K5YQ, 10 2015.

[102] MSDN. Conditional compilation in visual basic. http://bit.ly/1C5LYz9, 2015.

[103] MYERS, D., AND STOREY, M.-A. Using dynamic analysis to create trace-focused user interfaces for ides. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2010), FSE '10, ACM, pp. 367–368.

[104] NASEEM, R., MAQBOOL, O., AND MUHAMMAD, S. Improved similarity measures for software clustering. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering* (Washington, DC, USA, 2011), CSMR '11, IEEE Computer Society, pp. 45–54.

[105] PADMANABHAN, P., AND MADAPPA, S. Netflix queue: Data migration for a high volume web application. http://nflx.it/1Fk493Z. Job title: Architect & Architect at Netflix.

[106] QUEILLE, J.-P., VOIDROT, J.-F., WILDE, N., AND MUNRO, M. The impact analysis task in software maintenance: A model and a case study. In *Proceedings of the International Conference on Software Maintenance* (Washington, DC, USA, 1994), ICSM '94, IEEE Computer Society, pp. 234–242.

[107] RAHMAN, A. A. U., HELMS, E., WILLIAMS, L., AND PARNIN, C. Synthesizing continuous deployment practices used in software development. In *Proceedings of the 2015 Agile Conference* (Washington, DC, USA, 2015), AGILE '15, IEEE Computer Society, pp. 1–10.

[108] RAHMAN, M. T. Chrome architecture replication package. https://github.com/tajmilur-rahman/chrome-architecture, 11 2017.

[109] RAHMAN, M. T., QUEREL, L.-P., RIGBY, P. C., AND ADAMS, B. Feature toggles: Practitioner practices and a case study. In *Proceedings of the 13th International Conference on Mining Software Repositories* (New York, NY, USA, 2016), MSR '16, ACM, pp. 201–211.

[110] RAHMAN, M. T., AND RIGBY, P. C. Contrasting development and release stabilization work on the linux kernel. In *Intl. Workshop on Release Eng. 2014* (2014).

[111] RAHMAN, M. T., AND RIGBY, P. C. Release stabilization on linux and chrome. *IEEE Software 32*, 2 (Mar 2015), 81–88.

[112] RAYMOND, E. S. *The Cathedral and the Bazaar*. O'Reilly and Associates, 1999.

[113] RIGBY, P. C., AND ROBILLARD, M. P. Discovering essential code elements in informal documentation. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), ICSE '13, pp. 832–841.

[114] ROSSI, C. 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.

[115] RUBIN, J., CZARNECKI, K., AND CHECHIK, M. Cloned product variants: From ad-hoc to managed software product lines. *Int. J. Softw. Tools Technol. Transf. 17*, 5 (Oct. 2015), 627–646.

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

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

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

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

[120] SHIHAB, E., BIRD, C., AND ZIMMERMANN, T. The effect of branching strategies on software quality. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (New York, NY, USA, 2012), ESEM '12, ACM, pp. 301–310.

[121] SOCHOS, P., PHILIPPOW, I., AND RIEBISCH, M. Feature-oriented development of software product lines: Mapping feature models to the architecture. In *Object-Oriented and Internet-Based Technologies: 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World, Net. ObjectDays 2004 Erfurt, Germany, September 27–30, 2004 Proceedings* (2004), vol. 3263, Springer, p. 138.

[122] STROULIA, E., AND SYSTÄ, T. Dynamic analysis for reverse engineering and program understanding. *SIGAPP Appl. Comput. Rev. 10*, 1 (Apr. 2002), 8–17.

[123] THEREGISTER.CO.UK. Google's native client goes live in chrome. http://www.theregister.co.uk/2011/09/16/native_client_debuts_in_chrome/, 09 2011.

[124] TORVALDS, L. "Re: IRQF_DISABLED problem [maintaining status quo unless change is obviously better]". Linux Kernel Mailing List http://kerneltrap.org/mailarchive/linux-kernel/2007/7/26/122293, 2007.

[125] TSANTALIS, N., CHAIKALIS, T., AND CHATZIGEORGIOU, A. Jdeodorant: Identification and removal of type-checking bad smells. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on* (2008), IEEE, pp. 329–331.

[126] TSANTALIS, N., AND CHATZIGEORGIOU, A. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software 84*, 10 (2011), 1757–1782.

[127] TZERPOS, V., AND HOLT, R. C. Acdc: An algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)* (Washington, DC, USA, 2000), WCRE '00, IEEE Computer Society, pp. 258–.

[128] W CRESWELL, J. *Research design: Qualitative, quantitative, and mixed methods approaches*. SAGE Publications, Incorporated, 2009.

[129] WU, J., HASSAN, A. E., AND HOLT, R. C. Comparison of clustering algorithms in the context of software evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance* (Washington, DC, USA, 2005), ICSM '05, IEEE Computer Society, pp. 525–535.

[130] XIA, X., ZHOU, X., LO, D., AND ZHAO, X. An empirical study of bugs in software build systems. In *2013 13th International Conference on Quality Software* (July 2013), pp. 200–203.

[131] YIN, R. K. *Case Study Research: Design and Methods*, 3 ed., vol. 5 of *Applied Social Research Methods Series*. Sage Publications Inc., 2003.

Appendix A

Key-Words and Terms Used

A.1 Rapid Release

Software releases with shorter release cycles in a regular interval is usually known as a Rapid release. For the traditional longer version releases, the release length used to be longer, sometimes around a year. In the traditional release process developers used to be not committed to guarantee the release date and the release frequency. However, in rapid release model developers are committed to deliver the current features on the scheduled date because the the next release development will be started immediately.

A.2 Release Cycle

Release cycle is a cyclic process that includes development new features, merge and integration, testing and bug-fixing, and finally production launch.

A.3 Feature Toggle

Feature toggles are known as many other different terms such as Feature Flips, Feature Bits, Feature Flags or Simply Flags. It is a simple "IF" condition that blocks a piece of code from the execution at run-time of an application.

A.4 Feature Architectural View

A view point of the software architecture from the perspective of modules and the features coupled to the modules. The modular architecture represents the modules in a system. On the other hand the feature architecture represents the features and the relationships between them. The feature architectural view represents both modules and features at the same time along with their cross-cutting relationships.

A.5 Call Stack

A call stack is a stack that stores information about the active subroutines of a software system or just a software program. They are also known as program stack, execution stack, control stack, run-time stack, or machine stack.

A.6 Churn and Commit

Churn is a set of changes in one or multiple files that are submitted to the repository as a single change set. The submitting event of a churn to the repository is commonly known as a **Commit** in the version control systems.

A.7 Flaky Tests

Flaky tests are non-deterministic. Tests that do not result the same for the same piece code and the same set of inputs are usually known as flaky tests.

A.8 Crash Signature

Crash signature is a classifier that classifies subsequent crash reports caused by the similar error.