UNIVERSITÉ DE MONTRÉAL

MINING SOFTWARE REPOSITORIES FOR RELEASE ENGINEERS - EMPIRICAL STUDIES ON INTEGRATION AND INFRASTRUCTURE-AS-CODE

YUJUAN JIANG
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
AOÛT 2016

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

MINING SOFTWARE REPOSITORIES FOR RELEASE ENGINEERS - EMPIRICAL STUDIES ON INTEGRATION AND INFRASTRUCTURE-AS-CODE

présentée par: JIANG Yujuan
en vue de l'obtention du diplôme de: Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de:

M. GAGNON Michel, Ph. D., président
M. ADAMS Bram, Doctorat, membre et directeur de recherche
M. GUÉHÉNEUC Yann-Gaël, Doctorat, membre
Mme BAYSAL Olga, Ph. D., membre externe

DEDICATION

To my grandpa
Who has become a star in the heaven

ACKNOWLEDGMENTS

Many people I met, many things happened, are hidden as part of this thesis.

First of all, I want to thank my parents, for their endless and unconditional love. You respect my every single choice and impractical dream. You always hold the faith in me, even when I myself had lost it.

Exclusive thanks are dedicated to my supervisor, Dr.Bram Adams. Humble, enthusiastic and hard-working, for work and for life, you show me what should be like as a researcher and an individual person. You lead me through all the way up here, with enormous patience and kindness. In past four years, what I gained was not only professional skills, but also the will to face new challenges, the courage of learning from failures and the belief in hard working. Thank you Bram, I am so lucky to have you as my supervisor.

Thanks are also due to my all co-authors for their amazing collaborations: Prof. Daniel German and Prof. Foutse Khomh. Without you guys this thesis would not have been possible.

I would like to thank Dr.Michel Gagnon, Dr.Yann-Gaël Guéhéneuc, Dr.Olga Baysal for accepting my invitation to be jury members, Dr.Dominique Orban to be representative of my defense. I really appreciate your help.

Also, thanks are recorded to professors that gave me insightful advice in my academic life: Dr.Giuliano Antoniol, Dr.Peter Rigby, and Dr.Micheal Godfrey.

In addition, I want to thank my three best friends in Canada: Jiaqi, Qin and Xiaoxi, for your company, encouragement and watermelon in summer.

Last but not least, I wish to thank all colleagues and friends in our lab: Parastou, Latifa, Venera, Laleh, Soumaya, Aminata, Ons, Mahdis, Noopur, Le, Zephyrin, Wei, Francis, Ruben, Rodrigo... for all the inspiring talks, help and happiness. I will remember the days we spent together.

RÉSUMÉ

Release engineering (Releng) est le processus de la mise en production logicielle des contributions des développeurs en un produit intégré livré aux utilisateurs. Ce processus consiste des phases d'intégration, de construction et des tests, de déploiement et de livraison, pour finalement entrer au marché. Alors que, traditionnellement, la mise en production prend plusieurs mois pour livrer un produit logiciel complet aux clients, la mise en production moderne vise à apporter de la valeur au client plus rapidement, dans l'ordre des jours ou des semaines, pour recevoir de rétroaction utile plus rapidement et pour minimiser le temps perdu sur des fonctionnalités échouées.

De nos jours, une panoplie d'outils et de techniques a émergé pour soutenir la mise en production. Ils visent essentiellement à automatiser les phases dans le pipeline de la mise en production, ce qui réduit le travail manuel et rend le processus reproductible et rapide. Par exemple, Puppet est l'un des outils les plus populaires pour Infrastructure-as-Code (IaC), ce qui automatise le processus de mettre en place une nouvelle infrastructure (par exemple, une machine virtuelle ou un conteneur dans lequel une application peut être compilée, testée et déployée) selon des spécifications textuelles. IaC a évolué rapidement en raison de la croissance de l'infonuagique.

Cependant, de nombreux problèmes existent encore pour la mise en production. Par exemple, alors que de nombreux outils de la mise en production gagnent en popularité, le choix de la technique la plus appropriée exige des praticiens d'évaluer empiriquement la performance des techniques dans un contexte réaliste, avec des données représentatives. Pire encore, à un niveau plus haut, les ingénieurs de la mise en production doivent analyser le progrès de l'organisation dans chaque phase de la mise en production, afin de savoir si la prochaine date de sortie peut être respectée ou si des obstacles se produisent. De nouveau, il n'y a pas de méthode cohérente et établie pour ce faire.

Pour aider les praticiens à mieux analyser leur processus de la mise en production, nous explorons la façon selon laquelle la fouille de référentiels logiciels (Mining Software Repositories; MSR) est capable d'analyser le progrès d'une phase de la mise en production ou d'évaluer la performance d'un outil de mise en production. MSR agit sur les données stockées dans des référentiels de logiciels tels que les systèmes de gestion de versions, les référentiels de bogues ou des environnements de révision technique. Au lieu que les développeurs, les testeurs et les examinateurs utilisent ces référentiels juste pour enregistrer des données de développement (telles que les changements de code, rapports de bogues ou des révisions techniques),

MSR rend ces données actionnables en les analysant. Par exemple, en faisant l'extraction de l'information des changements de code source et de rapports de bogues, on peut recréer l'ensemble du processus de développement d'un projet ou de la mise en production. De nos jours, de nombreux référentiels logiciels à source libre sont disponibles au public, offrant des possibilités pour l'analyse empirique de la mise en production en utilisant des technologies MSR.

Dans cette thèse, on a fait des analyses MSR pour deux phases critiques de la mise en production, c.-à-d. l'intégration et le provisionnement de l'environnement d'un logiciel (avec IaC), sur plusieurs projets larges à source libre. Cette série d'études empiriques ciblait de comprendre le progrès du processus de la mise en production et d'évaluer la performance des outils de point. Nous nous sommes concentrés principalement sur ces deux phases parce qu'elles sont essentielles dans la mise en production, et un grand nombre de données est disponible pour elles.

D'abord, nous avons constaté que la révision technique et l'intégration de changements de code sont impactées par de différents facteurs. Nos résultats suggèrent que les développeurs réussissent à faire passer leurs contributions à travers la révision technique plus rapidement en changeant moins de sous-systèmes à la fois et de diviser une grande contribution en plusieurs contributions plus petites. En outre, les développeurs peuvent faire accepter leurs contributions plus facilement et plus rapidement en participant davantage dans la communauté de développeurs à source libre et d'acquérir plus d'expérience dans des sous-systèmes similaires.

Dans cette étude sur le noyau Linux, nous avons trouvé que l'un des défis majeurs de MSR dans le contexte de la mise en production logicielle est de relier les différents référentiels nécessaires. Par exemple, le noyau Linux ou le projet Apache HTTPD utilisent tous les deux des listes de diffusion pour effectuer le processus de révision technique. Les experts examinent des contributions par courriel, et un fil de courriels est utilisé pour ramasser toutes les différentes versions d'une contribution. Cependant, souvent un nouveau fil de discussion, avec sujet différent, est utilisé pour soumettre une nouvelle révision, ce qui signifie qu'aucun lien physique étroit n'existe entre toutes les révisions d'une contribution. En plus, les versions révisées d'une contribution non plus n'ont de lien physique avec la version acceptée dans le référentiel de gestion de versions, à moins qu'un identifiant de validation soit affiché dans un courriel. Surtout quand une contribution a été révisée plusieurs fois et a beaucoup évolué en comparaison avec la version initiale, le suivi à partir de sa toute première version est difficile à faire.

Nous avons proposé trois approches de différente granularité et de rigueur différente, dans le but de récupérer les liens physiques entre les révisions de contributions dans le même fil

de courriels. Dans notre étude, nous avons constaté que la technique au niveau des lignes individuelles fonctionne le mieux pour lier des contributions entre différents fils de courriels, tandis qu'une combinaison de cette approche avec celle à base de sommes de contrôle réalise la meilleure performance pour relier les contributions dans un fil de courriels avec la version finale dans le référentiel de gestion de versions. Être capable de reconstituer l'historique complet de contributions nous a permis d'analyser le progrès de la phase de révision du noyau Linux. Nous avons constaté que 25% des contributions acceptées prennent plus de quatre semaines pour leur révision technique.

Deuxièmement, pour évaluer la capacité de MSR pour analyser la performance des outils de mise en production, nous avons évalué dans un projet commercial une approche d'intégration hybride qui combine les techniques de branchement et de "feature toggles". Des branches permettent aux développeurs de travailler sur différentes fonctionnalités d'un système en parallèle, en isolation (sans impacter d'autres équipes), tandis qu'un feature toggle permet aux développeurs de travailler dans une branche sur différentes tâches en cachant des fonctionnalités sous développement avec des conditions "if" dans le code source. Au lieu de réviser leur processus d'intégration entièrement pour abandonner les branches et de passer aux feature toggles, l'approche hybride est un compromis qui tente de minimiser les risques des branches tout en profitant des avantages des feature toggles. Nous avons comparé la performance avant et après l'adoption de l'approche hybride, et avons constaté que cette structure hybride peut réduire l'effort d'intégration et améliorer la productivité. Par conséquent, l'approche hybride semble une pratique valable.

Dans la phase de provisionnement, nous nous sommes concentrés sur l'évaluation de l'utilisation et de l'effort requis pour des outils populaires de "Infrastructure-as-Code" (IaC), qui permettent de spécifier les requis d'environnement dans un format textuel. Nous avons étudié empiriquement les outils IaC dans OpenStack et MediaWiki, deux projets énormément larges qui ont adopté deux des langues IaC actuellement les plus populaires: Puppet et Chef. Tout d'abord, nous avons comparé l'effort de maintenance lié à IaC avec celui du codage et des tests. Nous avons constaté que le code IaC prend une partie importante du système dans les deux projets et change fréquemment, avec de grands changements de code. Les changements de code IaC sont étroitement couplés avec les changements de code source, ce qui implique que les changements de code source ou des tests nécessitent des changements complémentaires au code source IaC, et pourrait causer un effort plus large de maintenance et de gestion de complexité. Cependant, nous avons également observé un couplage léger avec des cas de test IaC et les données de provisionnement, qui sont de nouveaux types d'artéfacts dans le domaine de IaC. Par conséquent, IaC peut nécessiter plus d'effort que les ingénieurs expectent. D'autres études empiriques devraient être envisagées.

L'ingénierie de la mise en production moderne a développé rapidement, tandis que de nombreux nouvelles techniques et outils ont émergé pour le soutenir de différentes perspectives. Cependant, le manque de techniques pour comprendre le progrès des phases de la mise en production ou d'évaluer la performance d'outils de la mise en production rend le travail difficile pour les praticiens qui ont à maintenir la qualité de leur processus de mise en production. Dans cette thèse, nous avons mené une série d'études empiriques en utilisant des techniques de fouille des référentiels logiciels sur des données de larges projets à source libre, qui montrent que, malgré des défis, la technologie MSR peut aider les ingénieurs de la mise en production à mieux comprendre leur progrès et à évaluer le coût des outils et des activités de la mise en production. Nous sommes heureux de voir que notre travail a inspiré d'autres chercheurs pour analyser davantage le processus d'intégration, ainsi que la qualité du code IaC.

ABSTRACT

Release engineering (Releng) is the process of delivering integrated work from developers as a complete product to end users. This process comprises the phases of Integration, Building and Testing, Deployment and Release to finally reach the market. While traditional software engineering takes several months to deliver a complete software product to customers, modern Release engineering aims to bring value to customer more quickly, receive useful feedback faster, and reduce time wasted on unsuccessful features in development process.

A wealth of tools/techniques emerged to support Release engineering. They basically aim to automate phases in the Release engineering pipeline, reducing the manual labor, and making the procedure repeatable and fast. For example, Puppet is one of the most popular Infrastructure-as-Code (IaC) tools, which automates the process of setting up a new infrastructure (e.g., a virtual machine or a container in which an application can be compiled, tested and deployed) according to specifications. Infrastructure-as-Code has evolved rapidly due to the growth of cloud computing.

However, many problems also come along. For example, while many Release engineering tools gain popularity, choosing the most suitable technique requires practitioners to empirically evaluate the performance of the technique in a realistic setting, with data mimicking their own setup. Even worse, at a higher level, release engineers need to understand the progress of each release engineering phase, in order to know whether the next release deadline can be met or where bottlenecks occur. Again, they have no clear methodology to do this.

To help practitioners analyze their Release engineering process better, we explore the way of mining software repositories (MSR) on two critical phases of Releng of large open-source projects. Software repositories like version control systems, bug repositories or code reviewing environments, are used on a daily basis by developers, testers and reviewers to record information about the development process, such as code changes, bug reports or code reviews. By analyzing the data, one can recreate the process of how software is built and analyze how each phase of Releng applies in this project. Many repositories of open-source software projects are available publicly, which offers opportunities for empirical research of Release engineering.

Therefore, we conduct a series of empirical studies of mining software repositories of popular open-source software projects, to understand the progress of Release engineering and evaluate the performance of state-of-the-art tools. We mainly focus on two phases: Integration and Provisioning (Infrastructure-as-Code), because these two phases are most critical in Release

engineering and ample quantity data is available.

In our empirical study of the Integration process, we evaluate how well MSR techniques based on version control and review data explain the major factors impacting the probability and time taken for a patch to be successfully integrated into an upcoming release. We selected the Linux kernel, one of the most popular OSS projects having a long history and a strict integration hierarchy, as our case study. We collected data from reviewing and integration tools of the Linux kernel (mailing lists and Git respectively), and extracted characteristics covering six dimensions. Then, we built models with acceptance/time as output and analyzed which characteristics have impact on the reviewing and integration processes.

We found that reviewing and integration are impacted by different factors. Our findings suggest that developers manage to get their patch go through review phase faster by changing less subsystems at a time and splitting a large patch into multiple smaller patches. Also, developers can make patches accepted more easily and sooner by participating more in the community and gaining more experience in similar patches.

In this study on the Linux kernel, we found that one major challenge of MSR is to link different repositories. For example, the Linux kernel and Apache project both use mailing lists to perform the reviewing process. Contributors submit and maintainers review patches all by emails, where usually an email thread is used to collect all different versions of a patch. However, often a new email thread, with different subject, is being used to submit a new patch revision, which means that no strong physical links between all patch revisions exist. On top of that, the accepted patch also does not have a physical link to the resulting commit in the version control system, unless a commit identifier is posted in an email. Especially when a patch has been revised multiple times and evolved a lot from the original version, tracking its very first version is difficult.

We proposed three approaches of different granularity and strictness, aiming to recover the physical links between emails in the same thread. In the study, we found that a line-based technique works best to link emails between threads while the combination of line-based and checksum-based technique achieves the best performance for linking emails in a thread with the final, accepted commit.

Being able to reconstruct the full history of a patch allowed us to analyze the performance of the reviewing phase. We found that 25% of commits have a reviewing history longer than four weeks.

To evaluate the ability of MSR to analyze the performance of Releng tools, we evaluated a hybrid integration approach, which combines branching and toggling techniques together, in

a commercial project. Branching allows developers to work on different branches in parallel, while toggling enables developers on the same branch on different tasks. Instead of revising their whole integration process to drop branching and move to toggling, hybrid toggling is a compromise that tries to minimize the risks of branching while enjoy the benefits of toggling. We compared the performance before and after adopting hybrid toggling, and found that this hybrid structure can reduce integration effort and improve productivity. Hence, hybrid toggling seems a worthwhile practice.

In the Provisioning phase, we focus on evaluating the usage and effort of the popular tools used in modern Release engineering: Infrastructure-as-Code (IaC). We empirically studied IaC tools in OpenStack and MediaWiki, which have a huge code base and adopt two currently most popular IaC languages: Puppet and Chef. First, we study maintenance effort related to the regular development and testing process of OpenStack, then compare this to IaC-related effort in both case studies. We found that IaC code takes a large proportion in both projects and it changes frequently, with large churn size. Meanwhile, IaC code changes are tightly coupled with source code changes, which implies that changes to source or test code require accompanying changes to IaC, which might lead to increased complexity and maintenance effort. Furthermore, the most common reason for such coupling is "Integration of new modules or service". However, we also observed IaC code has light coupling with IaC test cases and test data, which are new kinds of artifacts in IaC domain. Hence, IaC may take more effort than engineers expect and further empirical studies should be considered.

Modern Release engineering has developed rapidly, while many new techniques/tools emerge to support it from different perspectives. However, lack of knowledge of the current Release engineering progress and performance of these techniques makes it difficult for practitioners to sustain high quality Releng approach in practice. In this thesis, we conducted a series of empirical studies of mining software repositories of large open-source projects, that show that, despite some challenges, MSR technology can help release engineers understand better the progress of and evaluate the cost of Release engineering tools and activities. We are glad to see our work has inspired other researchers to further analyze the integration process as well as the quality of IaC code.

TABLE OF CONTENT

LIST OF TABLES

LIST OF FIGURES

ABBREVNAME

Releng	Release engineering
MSR	Mining Software Repositories
IETF	Internet Engineering Task Force
OSI	Open Systems Interconnection
CI	Continuous Integration
DVCS	Distributed Version Control System
CVCS	Centralized Version Contol System
MM	superthread with Multiple patches across Multiple threads
MS	superthread with Multiple patches in a Single thread
SS	superthread with Single patch in a Single thread
LOC	number of Lines Of Code
IaC	Infrastructure-as-Code
OSS	Open Source Software

CHAPTER 1 INTRODUCTION

Release engineering (Releng) is the process of delivering integrated work from developers as a complete functional product to the end users as soon as possible [97]. It consists of a series of coordinated phases such as peer review, integration, building and testing, deployment, provisioning of infrastructure and finally release to market. Release engineering aims to automate the whole production pipeline across all phases to help release software products rapidly and consistently.

Modern software companies re-engineered their Releng process to reduce the time-to-market, and get feedback from end users faster. For example, Facebook and NetFlix can release a new version of the web service to end users twice per day, while Google Chrome and Mozilla Firefox browser push a new release every six weeks [32]. By leveraging new technologies and dedicated tools, faster Release engineering becomes possible and achieves great performance in practices. For example, a wealth of new environments have emerged that enable full integration of review , build and test process, and Infrastructure-as-Code (IaC) tools enable to instantiate the virtual machine or server environment of an application automatically.

However, while new Release engineering technology develops so fast, challenges come along. In industry, as more than 100 techniques/tools are available [17], it is difficult for practitioners to choose the best one to fit in their context. For example, to set up an application server, one can use Infrastructure tools like Puppet, but also existing declarative languages like Ruby. To deploy the server, one can choose a virtual machine with heavy-weight structure, or a Docker container, which is more flexible but has less stable tool chain. Especially for smaller companies and start-ups, it is important to help release engineers evaluate these tools, as migrating to new techniques costs more than what they can afford [139]. Furthermore, how can these new tools improve Release engineering process and do they really work as expected?

Apart from evaluating state-of-the-art tools, another important motivation on research is helping release engineers gain more sense about the progress of the Releng pipeline, or at least individual phases of it. For example, integration conflicts (i.e., the conflicts caused by different developers changing the same code block while working in parallel) are a serious problem that costs developers effort and time to fix. To reduce the number of conflicts, release engineers need help identifying bottlenecks in the integration process, after which they could think of solutions such as adopting a more efficient reviewing system or establishing concrete best practices for developers/maintainers. Since in school, release engineering is hardly taught [30]. Most software engineers lack a clear sense of Releng, and hence automated

techniques to understand the progress and problems in Releng phases is essential.

However, as release engineering is gaining more and more momentum, little research has been done to help release engineers evaluate the state-of-art tools as well as gain more insight into Releng progress. To explore these problems, we study the usage of mining software repositories of large open-source projects. These repositories, such as version control systems (Git, Subversion), bug repositories (Bugzilla, Jira) or revieiwing environments (Gerrit), are used on a daily basis to store all information of the development process during regular software and release engineering. In the last decade, the MSR research field has exploded with extraction and analysis of the data stored in these repositories to help address a wide variety of software engineering problems, from bug prediction to strategy-making. Our work explores the ability of MSR technology to address problems in the Releng domain, in particular to reconstruct the Releng process with these data and analyze how it progressed and what impact Releng tools could have on it. Our study focuses on two phases of Releng: Integration and Provisioning (Infrastructure-as-Code), as integration is the most complex phase of Releng and IaC is one of the most recent techniques. In this chapter, we will first introduce Release engineering in general and each individual phase, followed by our research hypothesis and that technologies we used. Finally, we discuss how our studies address the hypothesis.

1.1 Release Engineering (Releng) Pipeline

As shown in Figure 1.1, a typical Releng pipeline consists of a series of phases: Integration, Building, Testing, Provisioning (Infrastructure-as-Code), Deployment, and Release. We go through each phase of Release engineering in order, and talk about the current status and problems of each one.

Integration, one of the most complex phases of Releng pipeline, ensembles code from all developers in a coordinated way to form a complete and functional product. Modern version control systems enables different teams to work in parallel on separate tasks (features) by branching. Branching is a universal technique in software engineering that allows developers to work on a copy of the code base under version control. This allows different work to happen in parallel to improve productivity. Branches are isolated from each other, in order to prevent disturbing each other and improve development efficiency.

As an example, Figure 1.2 shows a company with a trunk (central) branch while team A and team B can work in their own copy of the trunk branch on different features respectively. After each feature is completed, developers will try to merge their branch back to the trunk, integrating new features into the code repository. Branches allow Team A and Team B to

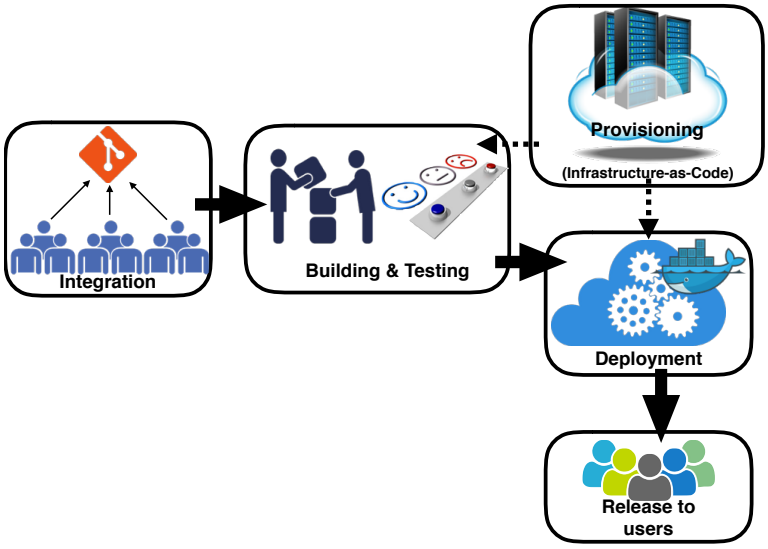


Figure 1.1 Overview of the whole process of Release engineering.

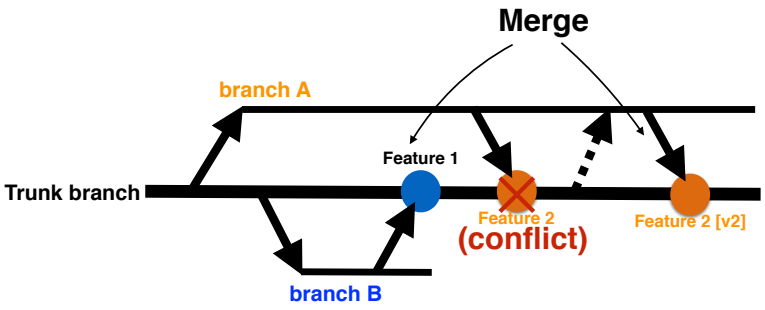


Figure 1.2 Example of branching structure.

work in parallel without disturbing each other.

In particular, branches try to minimize "Integration Conflicts", which are caused by developers in different branches changing the same code block. In the prior example (Figure 1.2), team A and team B start working on the same copy. However, when team A merged its feature 2 to trunk branch they found that some lines of code that they depended on had been changed by Team B in feature 1 (Integration conflict), so they were not able to merge their feature 2 back to trunk as they first need to synchronize from the trunk branch to get the latest version of the code (with feature 1's changes included), then fix the conflict and merge a new version of feature 2 back to trunk branch.

Many studies have been performed to analyze the integration process, such as to understand how branching techniques perform in practice [44] [59] [60] [146], prediction of integration conflicts and peer review process [48] [63] [134] [135] [136].

As software becomes larger and more complicated, code complexity increases and more people get involved. Hence, the Integration activity becomes more difficult and requires more effort. However, recent practices like Rapid Release require integration to be done even faster. It is critical to know how to make integration a more efficient and low-cost process.

The ***Building*** phase uses a build system to compile the source code and generate executables of the software product, taking as input specification files that record dependencies across files. Developers wrote their own ad hoc scripts to do this job, until 1977 Feldman published the first formal build tool "make" [76], which then became the primary build system of all Unix-like systems. Up until today, different types of Build systems have emerged. While "Make" is still the most traditional file-based build system, other technologies like task-based (e.g., Ant [2]) and lifecycle-based (e.g., Maven [13]) have become more popular in the meantime [113].

A lot of research has been done on Build systems [31] [114] [34] [85]. However, the build phase still has challenges during rapid release. As the release cycles has been shortened, a large number of code changes need to be built per day. This causes a lot of pressure on the build server and energy consumption.

Testing is a critical phase in Releng to guard software quality. Testing is divided in different stages starting with Continuous Integration (CI), which basically checks whether new code changes under review or committed to the version control system are compatible with the existing code base. Any new commit checked into the Version Control System (VCS) will trigger the CI tool (e.g., Jenkins), which compiles and tests this project partially or completely. CI mainly focuses on textual conflicts [63]. Other test types cover different level

(e.g., unit test and component test) and different functionalities such as regression testing that aims to make sure the whole product works [73], and performance testing to validate product meets required performance requirement [83].

Automation of Continuous Integration can help increase the speed of project development and expose potential conflicts as soon as possible. This approach has been widely adopted by IT companies like Google and Facebook.

Provisioning (Infrastructure-as-Code) is a new technology emerging with the development of cloud computing, aiming to fully automate the process of setting up a new environment (infrastructure), such as a virtual machine where applications can be compiled, tested and deployed, with required dependencies installed. For example, provisioning could generate a new image of virtual machine, install the required operating system, and deploy the product to the server.

This process can be repeated easily every time a new copy of this virtual machine is needed by re-executing the IaC tools. Developers can write infrastructure scripts with existing script languages such as Ruby, or with new declarative languages like Puppet and Chef. Hence, the ability to textually specify the infrastructure needs of a project led to the term Infrastructure-as-Code (IaC). As cloud computing rapidly develops, and techniques like virtual machine, and containers (e.g., Docker [6]) are becoming more and more popular, IaC becomes widely used in industry. IaC is not an independent phase in Release engineering but provides support to other phases.

IaC helps reduce manual and duplicate work, allowing the production process to progress without being disturbed by any side activities, so that the whole release engineering is accelerated.

Theoretically, IaC should be considered as normal source code, as all IaC files can be maintained, tested, and version controlled. However, as source code has many problems such as code clones, and increase of code complexity, IaC code could also face the same situation. Hence, although IaC aims to reduce manual effort and production cost, if it takes even more effort to maintain IaC code, it does not help practitioners as expected. Hence, we study the maintenance effort of IaC code in this thesis.

Deployment installs a version of the product in a staged directory, where it will be waiting to be delivered to end users. In this phase, the product is still under control of engineers, where the last phase of quality control will be performed. One typical practice is “canarying”, where the new product version will be pushed to a small group of data centers. If it works steadily in this “buffer period”, it will be pushed to all data centers [30].

Release is the phase where a product is made visible to end users. After the product is released to the market, engineers can collect the data about user feedback and product performance from the specific tracking tools and crash logs, to measure the satisfaction of users and quality of product. If any risk shows up, developers need to roll back to an older but stable version, or deliver an urgent new version with the bug fixed. Either way is complicated and costs effort.

However, new techniques rise to fix these problems. For example, "toggling" is a technique that uses "if" statements to control which features can be publicly seen or enable customers access to a specific service. Toggling allows long-term and consistent development process of a feature in rapid release circumstance, but at the same time it increases code complexity and introduces technical debt.

1.2 Mining Software Repositories (MSR)

Mining Software Repositories (MSR) is the technology that collects data from different data repositories, which track information of a software project's progress from different perspectives such as version control systems storing source code changes, reviewing systems storing code reviews and bug tracking systems storing bug reports. Researchers [165] [123] [48] have realized that by mining these data repositories, in particular by combining multiple repositories to each other, one can not only know the whole history of the development process, but also predict phenomena that might happen in the future. For example, bug prediction model is based on MSR technology. Simply speaking, researchers can analyze the bug tracking system to check which files were reported to have most bugs, then developers must pay more attention to these files when they make any changes.

Based on a specific research question, MSR techniques first need to identify the required data sources, extract the needed information, filter and link it (e.g., linking bug reports to the fixed files), then perform a specific analysis technique, like data mining or qualitative analysis. Finally, the results need to be interpreted to obtain actionable findings that help address the original research question.

As release engineering develops, phases generate substantial data, stored in repositories like version control systems, artifact servers or deployment logs, MSR techniques could be applicable as well to analyze context of release engineering. The main risk is that it is hard to link data between different repositories or to obtain access to repositories like deployment logs. Manually mapping between repositories takes time and may not be completely precise. In this thesis, we will study the progress of Releng and evaluate techniques/tool with MSR

technology.

1.3 Hypothesis of Thesis

Release engineering needs to deliver software products to the market faster. However, major challenges exist in each phase of Releng pipeline (as described in Section 1.1), such as integration conflicts. These challenges require better understanding of the whole Release engineering and correct evaluation of technologies. Hence, we aim to check how Mining Software Repositories can help address these challenges, leading to the following hypothesis of this thesis:

> *Mining the software repositories created during regular software and release engineering enables release engineers to understand the progress of a release engineering phase or evaluate the cost of release engineering tools and activities.*

To explore this hypothesis, this thesis follows a two-pronged approach. To study the usage of MSR to understand the progress of release engineering, we analyzed the factors impacting the probability and time needed for acceptance of a patch in a large, long-lived open-source project (Linux). To study how MSR can be used to evaluate Releng tools, we evaluate the usage of feature toggles as alternative for integration of new patches in an industrial system. We also evaluate the maintenance effort required for IaC in the large OpenStack and MediaWiki open-source systems. The following section details these contributions and their links with the research hypothesis.

1.4 Thesis contributions

1.4.1 Using MSR to Understand the Software Integration Process

We aim to understand the structure and performance of integration process by mining the integration and reviewing repositories of a large open-source software project. More specifically, we aim to analyze the probability of acceptance of patches submitted by contributors, and time it takes to review and integrate a patch.

We chose the Linux kernel as our case study as it is one of the most popular and influential open-source (OSS) projects. As the root of many operating system distributions, the Linux

kernel has a history of 25 years (first launched in 1991 as a personal project of Linus Torvalds), leading to a huge code base (over 19 million lines of code). It accepts contributions from more than 14,000 volunteers all over the world, and involves many commercial companies (e.g., Google, Apple) as well as freelancers. As a successful OSS project that integrates work from developers all over the world, the Linux kernel is a representative case study of an integration process.

The Linux kernel uses Git, a Distributed Version Control System, to manage integration based on branching. Linux has a rich integration hierarchy, where maintainers are responsible for subsystem repositories and Linux Torvalds is in charge of the central repository. Basically, each subsystem or module is maintained by separate maintainers. When a code patch passes the reviewing process, it will be integrated into the corresponding subsystem repository by its maintainer. If this patch works well and gets approved by this maintainer, it will likely have the chance to be integrated into the trunk branch and get into the core code base.

We collected 348,184 commits from the Linux integration system (Git) and emails from reviewing system (mailing lists) across eight years. In order to understand how a code patch is integrated into the Linux code base, namely from first being submitted by contributor and finally integrated into Linus Torvalds' code repository after being fully reviewed, we need to link the reviewing repositories and integration repositories together. This is a typical challenge with MSR technology.

With the complete integration process recovered, we were able to build statistical models to perform qualitative and quantitative analysis on the whole integration process. We found that reviewing and integration are impacted by different factors. For example, the number of affected subsystems, the number of contacted reviewers and submission time correlate the most with reviewing time, while patch maturity and prior subsystem churn play a major role in patch acceptance and integration time. In addition, developer experience is important in both the reviewing and integration process.

Our findings suggest that developers can get their patches go through reviewing faster by reducing the number of affected subsystems and reduce patch size. Furthermore, gaining more experience of development and more active participation in the community can help patches accepted sooner and more easily.

Our paper of the empirical study of the integration process has been cited 37 times by July 2016. This indicates that the integration process is an important topic for research and our

[0]including journal papers (one TSE, one TOCS, four EMSE and one IEEE Software), and conference papers (four ICSE, four MSR, four ICSME, four SANER, two SCAM, two ESEM), as well as other international symposiums, workshops and PhD theses.

findings provide a strong basis for further work.

1.4.2 Proposing Novel Linking Approaches to Link Repository Data

The Linux kernel has a rich hierarchy of repositories, yet reviews were email-based. In this low-tech reviewing system (mailing lists), developers submit code patches and maintainers review these patches all by emails. A physical link between emails across threads for the same patch or to an accepted commit is missing, especially when a patch needs to go through multiple revisions. So far no tools can recover such links in a low-tech reviewing environment. This makes it harder to (1) track the complete history of a code patch, (2) migrate the whole reviewing system to a modern structure, or (3) study the evolution history of code commits by researchers.

In order to study the progress of the integration process, we need to link emails about the same patch to each other, then to the commit accepted in the version control system. We proposed three different techniques with different granularity and strictness, to build a physical link for patches in emails and git commits. With quantitative analysis we found that the line-based technique is the best to link emails and finally accepted commit, while the combination of the line-based technique and the checksum-based technique has the best performance for linking patch versions between different email threads.

Our paper about linking approaches has been published in the conference Empirical Software Engineering and Measurement in 2014.

1.4.3 Using MSR to Evaluate Toggling Technique in Practice

Different to branching technique, toggling is a technique that enables developers to work on different features of the trunk branch. Toggling allows code to be compiled and tested in the most up-to-date project without the extra effort of keeping a branch synchronized with trunk or having to wait until a feature is completed. Indeed, each feature is guarded by a toggle (also called “switch”), which is actually an “if” statement. If the toggle is set to “false” (turned off), this feature will not be visible to the end-users. When a feature is ready to release, its toggle will be “turned on”. After the feature works steadily for a while, its toggle will be removed and the feature code will become permanent.

Many IT companies like Google and NetFlix adopt toggling in their development process. Toggling helps deliver new versions of a product rapidly. However, toggling gets rid of all feature branches in traditional way, and keeps only one trunk branch, which may be too risky for those companies wanting to adopt this new technique but lacking experience.

As a compromise, companies started to adopt a hybrid integration structure that combines toggling into a traditional branching structure, where branches are enhanced with toggling to help faster integration. Developers still work on branches but they can merge code changes back to trunk branch at any time, and do not necessarily wait until a feature is completed. We will evaluate whether this approach works in practice by comparing the performance of the integration process before and after adopting this technique, in order to help practitioners evaluate if this practice is worth trying.

Our case study is a real commercial software project in industry. This project started with a pure branching structure in May 2014, and gradually added toggles as of May 2015. We collected the data from their Git (development) and Gerrit (reviewing) repositories. To compare these two approaches, we used the KPIs (key performance indicators) proposed by managers of this company covering two dimensions "Integration effort" and "Productivity".

We found that this hybrid integration structure can take less integration effort to contribute a complete feature compared to pure branching, and abandon less commits. Toggling enables more smaller commits to be integrated into the product. This makes developers feel secure to submit their code changes soon and reduce the integration conflicts caused by working in isolation for long time.

According to our study, we found that a hybrid structure combining both branching and toggling achieves good performance compared to pure branching structure. Hence, we suggest those companies that hesitate to migrate branching to completely toggling to start with this hybrid structure and gradually migrate to pure toggling if everything keeps going well.

This paper will be submitted to IEEE software journal.

1.4.4 Using MSR to Understand the Infrastructure-as-Code Evolution Process

We conducted an empirical study of maintenance effort of IaC in the large open-source ecosystem OpenStack. Similar to the study of the Integration process, this empirical study is conducted on a major open-source project with substantial infrastructure needs/dependencies. To support this, OpenStack adopts two popular IaC languages, Puppet and Chef. In this thesis, we analyzed IaC code from two perspectives:

- Do IaC code changes co-evolve with source code changes?

  As the source code of a project keeps evolving and effort is needed to maintain it [69], we are interested in analyzing whether IaC is coupled with source code in terms of code changes, since this would also mean an increase of IaC code complexity and maintenance

effort. Hence, understanding whether IaC code changes are coupled with other code helps practitioners avoid possible trouble.

In our study, we collect all revisions of project files that ever existed in the version control system, and classify them into four different categories: Infrastructure, Production (source code), Build, and Test Code. We do a quantitative analysis of IaC code and use association rules to measure the coupling relationship between Infrastructure code with each of the other categories, trying to understand if any of them co-evolved.

We found that IaC code takes up a rather large proportion among all files, with 28% of IaC code files change monthly and with a large churn size. This indicates that it indeed takes effort to maintain IaC code. Furthermore, IaC code is tightly coupled with test and source code in terms of both code changes and ownership. Additionally, the most popular reasons for co-evolution are "integration of new modules" and "update configurations". All these findings reveal that IaC code deserves more attention and support, in order to reduce the impact of IaC code on other code files or vice versa.

Our paper that was published in the conference of Mining Software Repositories 2015, was the first to empirically study IaC code. We are honored to see there is already research built on it, like Sharma et al. [144], who studied the quality of IaC code.

- Does IaC-related Code Evolve Similarly As Production-related Code?

  In our prior study that analyzed how IaC code is co-changed with other files, we found that IaC code requires effort to maintain and IaC code changes are coupled tightly with source code changes. However, IaC code comes with related artifacts such as data files that are required for generating a new environment or tests that evaluate the correctness of the resulting infrastructure. Here, we conduct a more detailed empirical study to understand how IaC-related code evolves and whether this evolution is similar to the evolution of regular production source code files?

  Besides OpenStack, we selected another large open-source software as an additional case study. MediaWiki is a wiki application that also uses IaC techniques to manage its continuous integration and testing processes.

  We collect data from top repositories that contain the most IaC code in OpenStack and MediaWiki, and we classify all file revisions into two high-level dimensions: IaC-related files (including Infrastructure Code, Infrastructure Test, and Infrastructure Data files) and Production-related files (including Production Code, Build, and Test files). In this study, IaC data files involve database, configuration and multimedia resource files.

  The findings are consistent with prior work, e.g., IaC code changes frequently with large

churn size. Meanwhile, we also observed that IaC code files are lightly coupled with IaC data and test files. This might be risky for practitioners since the coupling relationship is not obvious. Similar to the prior study, we found that most of the co-evolution again was due to "integration of new component/service". We suggest that supporting tools are needed to manage IaC code and its related files.

This paper was invited to a special issue of journal Empirical Software Engineering, and got a major revision.

1.5 Plan of the Thesis

This thesis is structured as follows: Chapter 2 discusses prior research that is related to our work, as well as how they inspire our work. Chapter 3 introduces the overall structure of our empirical studies. Chapter 4, 5, 6, 7 and 8 present details of our empirical studies, followed by a general discussion (Chapter 9). Finally, we conclude our work and introduce future work possibilities in Chapter 10.

CHAPTER 2 LITERATURE REVIEW

Release engineering aims to deliver a high quality software product, which is an integration of the contributions from developers, to end users as soon as possible. Release engineering consists of the phases of "Software Integration", "Building", "Testing", "Provisioning", "Deployment", and "Release". In this section, we first talk about the related work on Release engineering in general, then go through the Release engineering process phase by phase. Especially in the sections of "Software Integration Process" and "Provisioning", we discuss how these works are different from our work.

2.1 Overview of Release Engineering (Releng)

Humble et al. [97] introduced the notion of continuous delivery and offer detailed guidelines for practitioners of how to build an automated and continuous pipeline, to deliver high quality products to customers as frequently as possible. They are also the first to introduce the definition of Infrastructure-as-Code (IaC). Ruhe et al. [141] introduced a systematic methodology of planning the release process of a product, as well as support tools required. The authors aim to recommend possible solutions and directions for practitioners.

Bass et al. [46] provide the first detailed guide to apply DevOps architecture in practice. DevOps is a practice that requires coordination of the development team (Dev) and IT operations team (Ops), instead of in traditional software engineering where development and operations are separated by a mental wall that slows down feedback from production flowing back to the developers. The authors talked about the impact of DevOps on each phase, and explained in details how DevOps is impacted by the architecture of the whole system. Releng forms the bridge between both camps, hence this book introduces the different practices in Releng.

Adams et al. [32] offer an overview of the whole process of release engineering: "Integration", "Continuous Integration (Building and Testing)", "Build System", "Provisioning", "Deployment", and "Release", and discuss the challenges that industry engineers are subject to (e.g., the release process of mobile platform and web service face different challenges), and the possibilities open to researchers (e.g., research is needed to predict build results, or analyze the differences between popular Infrastructure languages). More importantly, the researchers explain how modern release engineering influences mining software repository research methodology, and accordingly propose a list of advice for researchers who are inter-

ested in mining related data repositories, such as paying attention to complicated branching structures and build strategies. One of the main concerns expressed by this paper is the lack of empirical studies to help industry evaluate if their tools and techniques really work as well as they expect, and help smaller enterprises or start-ups to choose the best solution and tools to avoid wasting time and money. This thesis aims to conduct empirical studies of Release engineering by mining software repositories to address these concerns.

Three release engineers from representative industrial IT companies (Google, Facebook, and Mozilla) shared their experience about release engineering from an industry perspective [30]. To some extent they agree that deployment strategies like "Canarying (a staged release strategy)" being an important and universal techniques for projects to deliver a new release to end users incrementally, while release pipeline scalability is an important issue and mobile platforms are facing totally different challenges. Again, a lack of empirical validation of these claims is mentioned, i.e., is there really hard evidence that these techniques really work? Finally, the engineers are also concerned about the lack of professional education of release engineering in the academic field. Unfortunately, teaching requires a solid body of best practices and knowledge, which requires empirical validation.

Rahman et al. [130] performed a systematic analysis of 11 popular practices of continuous deployment that are commonly used in industry, to help practitioners understand better the continuous integration process. They found that it is necessary to automate the whole software deployment process with sound practices consistently, but companies tend to copy other companies' practices and lack of sense of Release engineering. For example, many companies adopting modern code review tool without knowing the motivation. This confirms our concern that current industry does not understand well the progress of Release engineering. In addition, in their study they did not mention Provisioning, which we analyzed in our work.

These papers and books confirm the importance of support for practitioners to understand the progress of Release engineering and evaluate state-of-the-art tools. Our research hypothesis addresses these two problems.

2.2 Software Integration Process

A lot of related research has been done about understanding, measuring, and improving the integration process in software engineering. Hence, we will introduce how the integration process works in practice and how researchers conducted experiments to understand the integration process from different perspectives.

Integration is based on branching, which allows developers to work in parallel on a copy of the

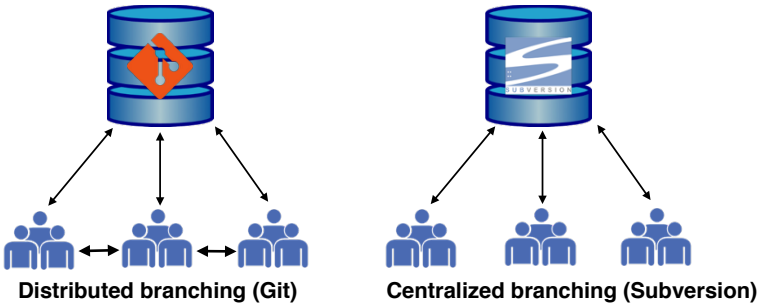


Figure 2.1 Example of Distributed branching (Git) and Centralized branching (Subversion).

central code base. Currently, the most popular branching structures are Distributed-based, which allows developers to communicate between each other and integrate each other's code changes, and Centralized branching, which only allows developers to submit code changes to the central code repository (see Figure 2.1). Often (e.g., in the Linux kernel project), a hybrid hierarchical structure is used, where different Git repositories are organized in layers, and each repository can only communicate with its direct, higher-level neighbour. Popular Distributed Version Control System (DVCS) are Git [9] and Mercurial [14], while the most popular Centralized Version Control System (CVCS) is Subversion [27].

Although branching enables developers to work in parallel, improving the speed of committing new code changes, being unaware of each other's work leads developers to change the same code block in a different way. Such conflicting changes only become apparent when integrating (merging) these changes back into a higher-level branch, at which time they require developers' effort to be fixed. Hence, predicting and avoiding integration conflicts is important to increase work efficiency and reduce waste of time and resources. Various researchers have studied the relationship between branching structure and software quality.

Brun et al. [63] studied collaborative development conflicts. They classified integration conflicts into "Textual conflicts" and "Higher order conflicts" ("Build conflict" or "Test conflicts"). They claimed that integration conflicts are frequent, persistent, and appear in different stages as well as forms. They also found textual conflicts to last as long as 9.8 days and involve 23.2 changesets to get fixed. This confirms the severity of integration conflicts. The authors claimed that 93% of integration conflicts are caused by unawareness of parallel changes and should have been avoided. Thus, they present an approach and develop a tool ("Crystal") for speculative analysis that can detect what is happening in other teams (potential conflict) before a developer pushes his change to a target repository, then offers corresponding guidelines about his available alternatives to resolve potential conflicts. This tool can help developers identify and resolve the integration conflicts at an early stage, before

actual merging of the conflicting changes.

Bird et al. [60] proposed a what-if analysis that allows to identify harmful, redundant branches to simplify the branching structure. Such simplification can avoid merge conflicts (incompatible changes in the two versions of the code base) that inflate integration time. In their case study on a large commercial system, such a simplification was able to reduce integration time by up to 9 days. Shihab et al. [146] did an empirical study on the Linux kernel project, in which they found how a mismatch between the branching structure and a company's organizational structure can increase post-release failure rate.

Perry et al. [128] conducted an observational study on a subsystem of Lucent Technology spanning 12 years. They found that parallel work cuts across common files and causes interfering changes. 12.5% of these changes are made by different developers to the same files. Longer parallel working time leads to interfering changes and frequent updates are necessary for coordinated changes.

These papers above confirm our concern that the improper branching structure and integration process may cause problems and affect software development process. Hence, understanding the progress of the integration process is critical.

The motivation of the software developers can also impact the project quality [151]. This motivation is different for different people and there is not a common, most important motivation factor [50]. However, it has been proven that low motivation may make the developers resist to build project of high quality [45] [160]. Fernandez et al. [77] concluded the key factors that affect the productivity of a project, including developers' experience, lack of experience of managers, the office environment, unpaid extra hours, moral factors, organization, annual training and schedule pressure.

Estublier et al. [75] conducted a study to explore the relationships between awareness, concurrent development process and software system. They found that the developers spend more effort on avoiding integration conflicts if they are aware of the influence of their code on other developers. However, Souza et al. [68] found that sometimes developers also rush their work into trunk to avoid being the one who would have to resolve the integration conflicts.

Al-Ani et al. [33] introduced three visualization tools implementing the *Continuous Coordination* paradigm and observed social behaviors of developers with these tools. They suggest that any tools supporting human social behaviors in software development processes should take into consideration cooperation between all stakeholders and project artifacts.

These works analyze the relationship between non-technical factors and integration productivity. Similar to their findings, we also found that developers' experience, participation in

community discussion and interaction with community members can affect the acceptance of contributions.

Compared to the empirical studies above, on the one hand, our work is consistent with their findings, such as smaller patches go through the reviewing phase faster and the interaction between contributors and community can help contributions get accepted faster. On the other hand, our work reveals new findings such as the characteristics of integration process in a low-tech reviewing environment. Furthermore, with a large code base and strict integration hierarchy, our case study of the Linux kernel contributes representative findings to this research area.

Rigby et al. [134] studied the peer reviewing process of a traditional inspection of a Lucent project and six open-source projects. They found that smaller changes take less review time and the time from the start of the review to the end of the review was in the order of weeks. Rigby et al. [136] also studied the reviewing practices in the Apache open-source system, and compared these to those of a commercial system. Between January 1997 and October 2005, the Apache mailing list contained 9,216 review email messages for 2,603 patches. They found that small, independent, complete patches are the most successful and that 44% of the submitted patches eventually were accepted. The low acceptance rate indicated the challenge of integration process.

Weissgerber et al. [156] studied contributions in two small open-source systems (196 and 1,628 patches respectively, changing 6% of all files in the systems), and also found that around 40% of the contributions are accepted. Similar to Rigby et al., they found that smaller patches have a higher probability of being accepted. Most of the patches are accepted in the repository within one week (61% even within three days). One quarter of the accepted patches took less than one day for reviewing and integration, half were accepted within one week, and one third took longer than two weeks. No relation between patch size and the time towards acceptance could be found.

Mcintosh et al. [115] recently studied the relationship between software quality and the quality of reviews in modern reviewing tools (Gerrit) to measure the impact of less thorough reviews in terms of number of post-release defects. The impact that they measured is due to a too large amount of freedom that reviewers have for choosing when to conduct the review via a modern reviewing web app like Gerrit. It is not clear whether low-tech reviewing environments based on emails are susceptible to the same issues.

Kononenko et al. [108] conducted an empirical study on the quality of the code reviewing process of Mozilla. They found that reviewers' prior experience may impact the review quality. Also, the characteristics of code patches such as patch size, the number of affected

files and the dependencies on surrounding code are related to bug-proneness of code review. Later, Kononenko et al. [107] performed a survey with developers of Mozilla, to investigate qualitatively the code reviewing process from the developers' point of view, such as the most influential factors and challenges they are subject to when conducting their code reviews. Consistent to our findings, they found that the patch size, developer experience and patch quality are considered by developers as influential factors for code review.

All research above is consistent with our findings, such as smaller code changes get accepted more easily and faster. In addition, the reviewing process plays an important role as a gatekeeper of the integration process. However, our analysis is the first to combine reviewing and integration processes together and address the problem of low-tech reviewing system and its impact on the software integration process.

2.3 Build Systems

As part of Release engineering and inspiration of our work about Infrastructure-as-Code, we also review the related work on Build system.

The first dedicated build system technology ("make") was developed by Feldman in 1977 [76]. Before "make" emerged, developers had to write their own scripts to configure dependencies and build a whole project. As more and more programming languages gain popularity, more and more build automation systems came into play. They are classified mainly into two categories: Make-based and Non-Make-Based tools. Make-based build tools are mainly for Unix-like systems. For example, GNU make and BSD make for Unix systems. Non-Make-Based tools cover a great variety of systems like Java-based (Apache Ant and Maven), Ruby-based (Rake), LISP-based (ASDF), C#-based (Cake), Facebook's Buck, Google's Bazel, and Microsoft's MSBuild.

Adams et al. [31] studied the evolution of the Linux kernel build system. They measured the change of size of build systems (SLOC), build artifacts and dependencies across releases. They found that the build systems evolve as the size, complexity, and maintenance effort increase.

McIntosh et al. [114] conducted a large empirical study on ten OSS projects of the maintenance effort and coupling of build system changes with source code changes. They found that build systems tend to evolve substantially and are coupled strongly with source code changes. This work motivated us to study the same phenomenon for Infrastructure-as-code, and we believe the methodologies that are used to study the (co-)evolution of the Build system can also be applied to IaC.

Al-Kofahi et al. [34] implemented a tool MkDiff to explore semantic changes in build makefiles recommending correct build process. Hardt et al. [85] provide a tool Formiga to support build system maintenance during evolution process.

As the Build system is outside scope of this thesis, we will not discuss this further here. Adams et al. [32] provide a more elaborate overview of related work.

2.4 Testing (Continuous Integration)

A Continuous Integration (CI) server monitors version control system, build, and test systems. The monitor will supervise the Version Control System (VCS), and arrange a series of follow-up activities for VCS events. In particular, when new commits are checked into the VCS, build and test systems will be triggered accordingly to automatically compile and test these new code changes.

Currently, CI tools are widely used such as Jenkins [11] (written in Java), Buildbot [3] (written in Python), Travis CI [28] (written in Ruby), Strider [26] (written in Javascript), and Go (written in Java). These CI tools have different structures and features. For example, Buildbot is server-slave-based structure, which is flexible to customize according to requirements, while many other CI tools have a fixed design that is less easy to extend. Strider even requires developers to play with the source code directly to configure. Go provides visualization functionalities that make a continuous delivery workflow visible. Companies must choose the suitable tools according to their needs. For example, the Eclipse community adopted Jenkins, while Mozilla and Google Chromium use Buildbot.

Vasilescu et al. [155] conducted an empirical study on the projects hosted by GitHub, analyzing the impact of adopting Continuous Integration servers on (1) Integration productivity (by the number of integrated pull requests), and (2) Software quality (by the number of defects). They claimed that CI use can improve productivity without harming software quality.

Continuous Integration requires to speed up the testing process. Research has been done to analyze how to optimize testing to meet the high speed requirement [163] [89]. More empirical research is needed to support these challenges.

Stahl et al. [149] conducted a literature review about Continuous Integration research and practices. They found that CI is implemented differently case by case. This finding confirms our motivation that practitioners should understand better the progress of Release engineering to find out the best solution for their own context.

2.5 Infrastructure-as-Code (IaC)

Infrastructure-as-code (IaC) is a new technique that automates provisioning of a new environment where applications can be deployed [10]. This automation make the whole process repeatable. Furthermore, IaC code can be considered as source code that can be version controlled, compiled, and tested. Popular IaC tools are Chef [150], Puppet [154], CFEngine [4] and Ansible [1].

The term was first introduced in the 2010 book by Humble et al. [97], while the technology goes back to at least 1993, when cfengine emerged as an alternative for basic bash scripting. The surging interest in DevOps and release engineering has sparked development of newer languages like Puppet (2005) and Chef (2009), followed more recently by Salt (2011) and Ansible (2012).

Morris [122] introduced the Infrastructure-as-Code and the benefits that IaC brings. More importantly he provided specific guidelines for practitioners about applying IaC in a practical development process.

Recently, motivated by our research about Infrastructure-as-Code (IaC), Sharma et al. [144] conducted a study of configuration code quality, trying to figure out whether the code smell types of IaC code are similar to that of traditional source code. This study is based on 4,621 repositories containing IaC code. The researchers classify configuration code smells into two main categories ("Implementation" and "Design" Configuration Smells) covering 24 types totally. They find that configuration code smells tend to co-occur within or across different types, and there is no negative correlation between configuration smells and size of a configuration management system. This paper looks into the IaC code quality while ours focuses on the evolution of IaC code. However, we are very glad to see that our work on Infrastructure-as-Code has drawn more attention on this research area and could inspire other researchers.

As one of the new technologies coming along with cloud-computing, little research has been done about Infrastructure-as-Code (IaC). This confirms that more empirical research must be done to help practitioners understand this new technique better and help them choose the right tool and make the right decision.

2.6 (Continuous) Deployment

Dearle et al. [69] introduced the definition of software deployment. He studied six different deployment technologies and approaches including Java Beans, Linux, Ṅet, OMG, Service-

oriented Computing Paradigm, and Virtualisation. He discussed the characteristics and problems of each technology, explaining the variety and complexity of software deployment process. The author also exposed the issues of developing a software deployment methodology and potential solutions, such as strengthening security of communication in the distributed deployment process. In addition, mobile platform and sensor-net deployments also deserve more attention from the academic side.

Rodriguez et al. [139] conducted a literature review about continuous deployment, mainly covering ten themes such as Continuous testing and Post-deployment activities. They addressed the challenges of continuous deployment of transforming cost and increase of quality assurance. They also claimed that the current status of research about Continuous Delivery is high-relevant but low-rigor, and more rigorous research must be performed. The exposed ten themes also reflect the research gaps that call for more empirical studies.

Continuous Deployment is currently a prevalent methodology in industry, which aims to shorten the time of development and release, reduce conflicts caused by working in isolation for a long time, deliver to end uses faster and get feedback from the market sooner. Many commercial and open-source communities have adopted Continuous Deployment/Release in practice. However, do faster release really bring so many benefits? Is there potential risk that has not been revealed? Mäntylä et al. [112] conducted an empirical case study on the Mozilla project to figure out the relationship between rapid release (RR) and software quality, in terms of the change of software testing and building. It turns out that rapid release reduces the number of builds and platforms tested per day. Furthermore, the number of test executions increases but the testing scope decreases, and the software quality is not proven to be significantly improved nor degraded.

2.7 Release

The final step in Release engineering, Release, makes the product visible to all end users. Due to a highly-competitive market, the Release process tends to become faster and faster. Many software communities like Mozilla and Google adopt Rapid Release (RR) practice and deliver new versions of products much sooner than before.

Baysal et al. [47] conducted an empirical study on two open-source software products: Google Chrome and Firefox. They mined the release historical information and compared these two browsers from three dimensions including software quality, user experience and popularity. They found that the two browsers presented different characteristics. As an established browser with longer history, Firefox has a longer release lifespan and bugs are fixed sooner

than Chrome. On the contrary, Chrome has a faster release but longer bug-fix cycle. In addition, users of Chrome prefer to update to the new version more frequently than Firefox users. Baysal et al. [48] also analyzed the impact of adopting Rapid Release practice on the code revieiwng process. They found that Rapid Release helps accelerate the reviewing process, and the number of both accepted and abandoned patches increases. As more contributions are accepted, problem like alienating valuable contributors by not responding in time may happen. Hence, support for positive experience is needed from the review process.

Khomh et al. [103] also empirically studied Mozilla Firefox, but they focused on the impact of rapid release on software quality before and after adopting rapid release approach. By analyzing the number of post-release bugs, user experience information, crash logs and bug tracking system logs, they found that rapid release can help expose/fix failures faster, and assist users to update applications faster.

Rapid Release accelerates the release cycle while potentially putting more pressure on the development process. The pressure of completing a feature may result in defects. Hence, quality assurance bears new challenges in the Rapid Release era. As one of the solutions, "Toggling" can help guard software quality from imperfect features. "Toggling" uses conditional statements ("toggles" or "switches") to control whether a feature should be visible to end users. Invalid code can exist under protection of turned-off toggles. Once such feature is stable in a couple of releases, the conditional code will be removed and the feature will become permanent.

Rahman et al. [132] studied the evolution and practical performance of toggles in Google Chrome. They found that toggling helps keep the development of large features in Rapid Release context consistent and reduces Release pressure, but at the same time introduces technical debt and increases code complexity. Toggles can be used in different way. In their work, toggles are used to test and release, while in our work about toggling, toggles are more used to improve integration efficiency. We explored the functionality of toggles from different perspectives.

2.8 Mining Software Repositories (MSR)

With automatic tools used in software engineering, data of development process and other production behavior is stored in different repositories of these tools, such as version control systems storing source code changes, reviewing systems storing code reviews and bug tracking systems storing bug reports. As Release engineering developed, ample data source are available for researchers to conduct related analysis.

Hassan et al. [86] discussed the history of MSR and how it had been adopted in empirical research such as to help understand software systems, predict and identify bugs, and improve the user experience. Also, MSR has been facing challenges like exploring non-structured data and linking data between repositories. Hassan et al. [87] also coined the name Software Intelligence (SI) as a future vision of MSR that can help practitioners with the decision-making process. Bird et al. [58] offered guidelines and discussed popular practices of analyzing data to help practitioners analyze software data from different resources.

As MSR develops, there are already hundreds of papers about MSR, and MSR has its own conference [15]. Here, we will discuss some of them, for other work, we refer elsewhere [86] and book [58], [88].

Mining Software Repository (MSR) technology is widely used in software engineering to do a variety of analysis such as development-related behavior, defect prediction, code change recommendation, and energy consumption.

Zimmerman et al. [165] empirically analyzed the progress of parallel development by mining CVS repositories of four large OSS software projects: Gcc, Jboss, Jedit, and Python. They talked about the limitation of mining CVS data such as invalid data by manual configuration. Tian et al. [152] propose a classification model that takes advantage of both textual and code specific features to identify critical bug-fixes out of all submitted commits, instead of manual diagnose is that may cause many missed data. Moura et al. [123] mined 371 energy-aware commits from 317 non-trivial applications on GitHub to qualitatively analyze how developers apply energy-saving solution in practice. They found that different energy-saving solution may impact on application functionality and it lacks of high-level API of energy solution for developers.

While MSR technology has been used to address challenges traditional software engineering was subject to, we use MSR to help practitioners gain a better sense of the progress of Release engineering phases and to evaluate the state-of-the-art techniques/tools.

One of the major challenges of MSR is to link data between different repositories to observe the overall process of a project.

Bird et al. [56] investigated the impact of data bias on bug prediction models. They claimed the concern that data bias may impact research. Data bias is that distribution of data is not fully balanced and random. For example, it is likely that not every developer submits bug report to quality assurance system regularly. Only developers in charge of a more fragile component tend to have problems. One cannot simply conclude that the number of bug report reflects the quality of the whole software project. In their validation process, they

found out a major challenge is to link commit with corresponding bug report. People can only informally identify the counterpart by manually analyzing log message. This results in partially missing data and causes bias in dataset.

To overcome the problem of missing linkage that caused data bias, tools are implemented to explore relationship between different data resources. Hipikat [19] finds links between different types of software artifacts based on heuristics and offers recommendation for developers. LINKSTER [42] inspects and integrates data from multiple sources by analyzing annotation of bug-tracking system and version control system. However, no tool cares about linkage in low-tech reviewing environment.

Based on our literature review, we discussed the different problems that exist in Release engineering and that state-of-the-art techniques/tools are not able to help with. Hence, our research aim to understand Release engineering progress and evaluate popular Releng tools by mining software repositories to help practitioners improve their development efficiency and productivity. In this thesis, we mainly focus on two phases of Release engineering, i.e., "Integration" and "Infrastructure-as-Code", where ample repository data is available for empirical research using MSR methodologies.

2.9 Insights of State of the Art

We conducted a thorough literature review on current status of the release engineering research field, and we found that there exists potentially a large number of possibilities for research, especially for "integration" and "provisioning" phases of release engineering. This is because "integration" is the most fundamental and complicated phase, while "provisioning" is one of the most recent practices. However, little research on them has been done. Thus, we focused on these two phases and performed a series of empirical analysis, to help practitioners gain more sense about release engineering. It turns out our findings are consistent with existing research. In addition, we provide novel advice for practitioners in case they need support when applying new techniques to re-engineer their release engineering.

CHAPTER 3 RESEARCH PROCESS AND ORGANIZATION OF THE THESIS

This chapter presents the methodology and the structure of this dissertation. This thesis focuses on "Integration" and "Provisioning (Infrastructure-as-Code)", two of the most important parts of Releng pipeline, and for which extensive data is readily available. Our work tries to help practitioners understand the progress of Release engineering and evaluate state-of-the-art tools/techniques by mining software repositories.

Chapter 4 and Chapter 5 analyze the performance of the Review and Integration in the Linux kernel project, while Chapter 6 studies the performance of a new integration technique (Toggling). Chapter 7 and Chapter 8 investigate the characteristics of IaC code and the effort needed to maintain IaC code.

3.1 Understanding the Progress of the Software Integration Process

Integration is a complex phase in the Release engineering pipeline. An efficient integration process requires understanding what other teams are working on and what are the characteristics of successful code contributions. To help practitioners understand better the internals of the integration process, we empirically study the integration process of the Linux kernel, one of the most popular open source software projects with a strict integration structure and a rich hierarchy of repositories. We collect data from reviewing system (mailing lists) and version control system (Git). One of the typical challenges of MSR technology is linking data from different repositories. In this case study, we need to figure out how to link different revisions of patches and accepted commits to each other to recover the whole integration process.

3.1.1 Empirical Study of Large Software Integration Process

In Chapter 4 we conduct an empirical study on the Linux kernel to understand the integration process in practice. We study the Linux kernel project as its large code base and the strict integration hierarchy provide a realistic case study: as one of the most successful large open-source software (OSS) project, the Linux kernel has a long history (as of 1991), which indicates a large amount of available data to study, and it uses Git, the most popular Distributed Version Control System (DVCS). Between 8,000 and 12,000 patches going into each recent kernel release results in more than 15M lines of code (LOC). We studied 348,184 commits

totally in our work.

In this study, we aim to understand what factors play a major role in the integration process in large OSS project. This will help developers know better how to contribute their code and get it accepted more successfully, as well as help maintainers of the Linux kernel to identify the valuable patches within the overwhelming submissions per day. Improving integration efficiency will decrease the cost of the integration process.

3.1.2 Tools Needed to Support Integration Process in Low-tech Reviewing Environment

The Linux kernel project uses mailing lists to perform reviewing process. Compared to modern reviewing systems such as Gerrit, where the whole process of code review can be tracked and stored, we defined this email-based review system, where physical links are missing between emails talking about the same patch, as "low-tech" review system. In this low-tech reviewing environment it is difficult to trace back the original rationale of a code change especially after it has evolved through multiple revisions.

In our prior study (3.1.1), we proposed a checksum-based algorithm to link patches in reviewing emails to the final commit in the version control system. However, we wonder if there are other more efficient approaches to do this job. To select the best linking approach, we designed and evaluated three different algorithms/tools from different granularity/restrictions to compare their performance in terms of linking different revisions of patches to each other, or patches & accepted commits. The three different approaches are: clone-based with CCFinder tool (token level, least strict), plus-minus-line-based technique (line level, medium strict), and checksum-based technique (chunk level, the most strict). More details will be discussed in Chapter 5.

3.2 Evaluating a Hybrid Integration Process (Branching & Toggling)

Toggling enables developers to work on different tasks (features) on the same branch. However, it might be too risky for companies to get rid of a traditional branching structure immediately.

Company S launched product P (anonymous names) in May 2014. It started with a branching structure for integration. As of May 2015, this company started adopting toggles in project P. In contrast to pure toggling, this company did not migrate to one single branch. Instead, it keeps all branches and the central trunk branch, but add toggles in their source code in order to improve productivity. If this hybrid structure can achieve a better performance than

branching, other companies can follow the same practice to minimize the potential risk of migrating to a new integration technique too soon.

We extracted data about product P from Git (integration) and Gerrit (review) repositories, trying to measure the performance of this hybrid integration structure. In our empirical study of Chapter 6, we will compare integration with and without adopting toggling in practice from two perspectives: *Integration effort* and *Productivity*.

3.3 Evaluating the Maintenance Effort of Infrastructure-as-Code (IaC)

Infrastructure-as-Code (IaC) is a new technique that aims to save effort compared to manual provisioning of an infrastructure (e.g., a virtual machine where applications can be compiled and deployed). IaC code should theoretically be considered as normal source code that can be tested, built, and version controlled. However, as it takes effort to maintain normal source code, IaC code may have the same issue. If it takes more effort to maintain IaC code than that it saves, this technique does not improve Release engineering. Hence, we conduct empirical studies to evaluate how much effort is required to maintain IaC code and how IaC code evolves.

3.3.1 Co-evolution of IaC with Production Code

To evaluate IaC in terms of maintenance effort, we first conducted an empirical study of OpenStack, a cloud-based open source software system, which adopts two popular IaC tools in its development process: Puppet and Chef. We collect data from 265 Git repositories that contain IaC code and classified files into five categories: *IaC files*, *Production files (source code)*, *Build files*, and *Test files* (*"Other" files* were disserted). We first analyzed the characteristics of IaC code files in OpenStack, such as the proportion and size of IaC code files. Then, we measure its maintenance effort by quantitatively computing how frequently they change and the churn size. Furthermore, we analyze if IaC code changes are coupled with other production code changes as coupling with other file categories may cause even more maintenance effort, presented in Chapter 7.

3.3.2 Understanding the Evolution of Infrastructure-as-Code (IaC)

Our prior study about IaC code confirmed our concern that IaC code takes developers effort to maintain and deserves much more attention from both industry and research sides. Hence, we performed a deeper empirical study to study IaC code. We select two case studies, OpenStack and MediaWiki, which adopt IaC in a different way. OpenStack uses its IaC to manage its

continuous integration and testing processes while MediaWiki uses it in regular development process and to deploy production servers. We collect data from top repositories that contain most IaC code and classify all file revisions into more detailed categories: IaC-related files (IaC code, IaC test and IaC data files) and Production-related files (Source code, Build, and Test files). We aim to understand if the evolution of IaC-related files has the same trend as that of Production-related files. This is the subject of Chapter 8.

3.4 Overview of the Thesis Organization

This thesis aims to explore the modern release engineering process from two perspectives by using MSR (Mining Software Repositories): on the one hand we try to understand characteristics of the current status of release engineering, while on the other hand we try to evaluate state-of-the-art techniques and practices. As the whole release engineering process covers multiple phases and it is hard to get data for certain phases, we focus on two phases of release engineering: "integration" and "provisioning (Infrastructure-as-Code)". We conducted an empirical study to understand the integration process of the Linux kernel project in Chapter 4. To support later research about the integration process based on such "low-tech" environments, we proposed linking techniques to recover the whole integration process in Chapter 5. We also evaluate the performance of the popular "feature toggle" integration technique in an industrial environment (Chapter 6). In the context of "provisioning", we measure the potential maintenance effort it might bring to industry and explore its evolution process in Chapter 7 and Chapter 8.

CHAPTER 4 ARTICLE 1: WILL MY PATCH MAKE IT? AND HOW FAST

Yujuan Jiang, Bram Adams, Daniel German

Abstract

The Linux kernel follows an extremely distributed reviewing and integration process supported by 130 developer mailing lists and a hierarchy of dozens of Git repositories for version control. Since not every patch can make it and of those that do, some patches require a lot more reviewing and integration effort than others, developers, reviewers and integrators need support for estimating which patches are worthwhile to spend effort on and which ones do not stand a chance. This paper cross-links and analyzes eight years of patch reviews from the kernel mailing lists and committed patches from the Git repository to understand which patches are accepted and how long it takes those patches to get to the end user. We found that 33% of the patches makes it into a Linux release, and that most of them need 3 to 6 months for this. Furthermore, that patches developed by more experienced developers are more easily accepted and faster reviewed and integrated. Additionally, reviewing time is impacted by submission time, the number of affected subsystems by the patch and the number of requested reviewers.[1]

4.1 Introduction

Integration of code changes into a project's main repository is an open source developer's ultimate goal, since it marks the first step towards inclusion in an official product release. An open source project like the Linux kernel, for example, integrates between 8,000 and 12,000 patches in a new release, contributed by more than 1,000 developers [66]. Those patches only represent the "lucky few". Studies on Apache and other open source systems have shown how only 40% of the patches considered for integration eventually succeed [48, 135, 156].

One of the major reasons for the relatively low success rate of integration is the complexity of this process. The patches first need to pass a gate-keeper who performs a review of the code [48, 120, 136], before the code is merged by an integrator (e.g., release engineer) into the corresponding branch of the open source project [60, 63, 146]. Code reviews fail when a patch does not implement a relevant, working feature or bug fix, or when the project's

[1] This paper has been published in the conference of Mining Software Repositories 2015.

development guidelines are not followed [65]. The actual integration (merging) fails when the patch interacts incorrectly with other patches or the merging process creates too many merge conflicts [109]. In case of integration issues, the developer needs to go back to the drawing board and try to integrate the code again. In the worst case, a patch will be rejected time and time again until the developer eventually gives up.

As a result, the integration process looks like a black box to most developers, with unpredictable outcome. Everyone knows the stories of disgruntled developers, even experienced ones, whose changes did not make it after putting months of work into them (e.g., [35, 118]). Even major projects like the Google Android mobile platform have problems getting their Linux kernel modifications integrated into the official kernel version [109]. Yet, determining up front whether a patch will make it, and how long it will take, is a grey area. Research on code reviews has shown how small patches [136, 156] sent by experienced developers [48] are more likely to be accepted by the reviewers, but it is not clear if these characteristics play the same role during the actual integration of the patch with other patches. Similarly, the impact of these characteristics on the time it takes to get a patch into a release is unclear.

This paper studies the relation of patch characteristics with (1) the probability of acceptance into an official release and (2) the time between submitting a patch for review and acceptance. We also analyze if these relations change over time. Our empirical analysis is based on eight years of patch review data and version control data from the Linux kernel project, which is a 20 year-old, popular open source system containing more than 15 MLOC of source code. We address the following research questions:

RQ1) *What percentage of submitted patches has been integrated successfully, and how much time did it take?*

Around 33% of patches are accepted. Reviewing time has been dropping down to 1–3 months, while integration time steadily has been increasing towards 1–3 months, bringing the total time to 3–6 months.

RQ2) *What kind of patch is accepted more likely?*

Developer experience, patch maturity and prior subsystem churn play a major role in patch acceptance, while patch characteristics and submission time do not.

RQ3) *What kind of patch is accepted faster?*

Reviewing time is impacted by submission time, the number of affected subsystems, the number of suggested reviewers and developer experience, while integration time is impacted by the same attributes as patch acceptance.

First, we provide background about the Linux kernel integration process (Section 4.2), followed by an explanation of our case study methodology (Section 5.3). Section 4.4 presents the results of our case study, followed by a discussion of threats to validity (Section 8.5). We finish the paper with related work (Section 5.6) and the conclusion (Section 5.7).

4.2 Background

The Linux kernel open source project started out as a one-man project by Linus Torvalds in 1991, but quickly exploded into one of the hallmark projects of open source development. Since early on, kernel development is managed by a strict hierarchy of experienced kernel developers under the leadership of Linus Torvalds, who has the final decision about incorporating a source code patch into the kernel. Figure 4.1 illustrates the lifecycle of a kernel patch [65].

A developer first needs to send a request for comments (RFC) to a kernel subsystem's mailing list to get input on a new idea for a feature or bug fix. After fleshing out the design, the developer implements it and sends the resulting patch or patch set (series of collaborating patches) to the subsystem's mailing list and/or the global Linux Kernel Mailing List (LKML). Through email discussion, the subsystem maintainer and other experts review the patch. The developer then needs to incorporate any feedback and re-submit his patch (set), otherwise the patch does not get through the reviewing stage.

Once all reviewers are happy, the subsystem maintainer commits the final patch to his Git repository. Other maintainers, developers and beta-testers tracking the maintainer's repository now become aware of the patch and can provide additional reviews. This will also expose integration conflicts, i.e., other patches that break because of the new patch. Again, the developer needs to act on these conflicts to avoid not getting through integration. If the kernel maintainers are sufficiently confident about the patch, Linus Torvalds might consider incorporating it into the official Linux kernel release. This only happens during a release's "merge window", a period of roughly two weeks following the previous release. Afterwards, small bug fixes are still possible until the next release occurs (every 2 or 3 months [66]).

This paper uses the terminology of Figure 4.1 to denote the duration of each major phase. **Reviewing time** is the time from a developer's patch submission until the patch's commit by a subsystem maintainer. **Integration time** is the time from a patch's commit by a

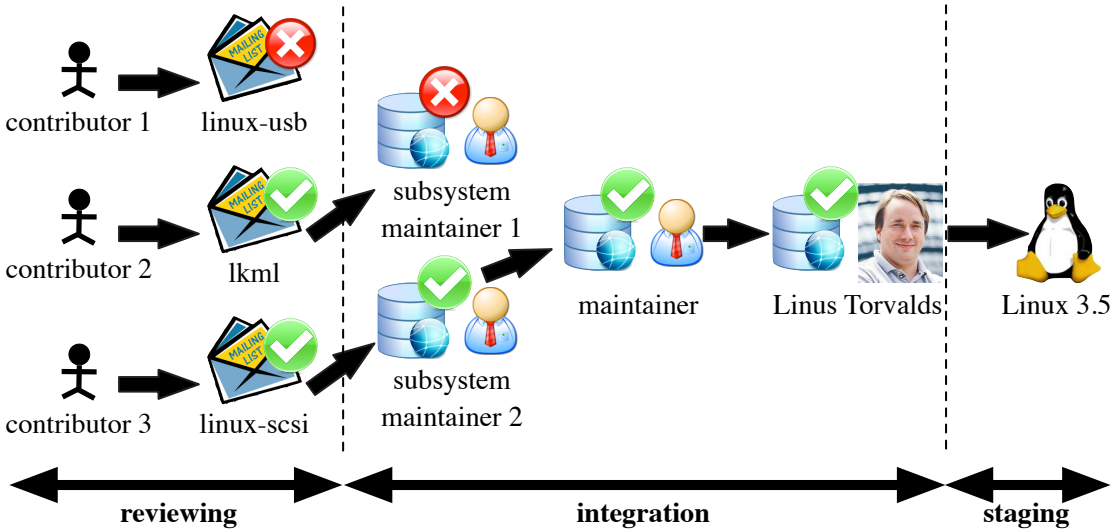


Figure 4.1 Linux kernel development process. Contributor 1's patch is rejected during reviewing, and contributor 2's patch is rejected during integration, but contributor 3's patch makes it into the next Linux release (version 3.5).

subsystem maintainer until its merge into Linus Torvalds' repository. Finally, **staging time** is the time from a patch's merge into Linus Torvalds' repository until the next kernel release.

4.3 Methodology

In order to address our three research questions, we studied the acceptance rate of patches submitted by email, as well as their reviewing, integration and staging time. We now explain the different steps used for our study.

4.3.1 Data Extraction

Since the Linux kernel integration activities (Figure 4.1) are spread out across mailing lists and Git repositories, we need to mine both data sources, then try to match the patches inside the emails to commits inside the repositories.

There is one global mailing list (LKML), and 130 more specialized kernel-related lists. These mailing lists are archived online [74] in the form of textual mbox files. We obtained access to the mbox files of 2005 until 2012, then used the MailMiner tool [54] to process these files into a relational database, with tables containing the email messages and their metadata. We did not analyze email attachments, since the Linux kernel developer guidelines dictate that all patches are inlined into the email body(i.e., attachments are not reviewed). Unfortunately, we were not able to find reliable heuristics that link threads related to (different versions

of) the same patch to each other, since no official guidelines exist regarding the title of such email threads.

The source code repository data is better structured. Since the 2.6.12 kernel (June 26, 2005), the kernel uses the Git distributed source control system where each developer and maintainer has a copy of the whole project repository. Since we only need to analyze which patches make it into an official release, we only have to clone and mine Linus Torvalds' repository [153]. This repository contains all commit information of accepted patches from June 26, 2005 to December 31, 2012, including information like the original commit date by a subsystem maintainer and the merge date into Linus Torvalds' repository. Note that we do not have information about patches that made it to a maintainer's repository, but never to Linus Torvalds' repository (the second example in Figure 4.1).

4.3.2 Linking the Patches in Emails to Git Commits

To obtain information on which patches are accepted by Linus Torvalds, we had to link the patches in the mailing list emails to the commits in Linus Torvalds' Git repository. Similar to Bird et al. [57], we did not directly link a full patch to a Git commit, because of "cherry-picking". This is a common integration activity where an integrator only picks the interesting parts of a patch and ignores the rest. Large patches risk not being merged completely, or maybe not in one Git commit. Hence, we split a patch into one or more chunks, with each chunk containing all changes of the patch to one file. If a patch undergoes multiple versions in one email thread (we cannot track versions across threads), we tried to link each patch version to the Git repository.

The actual linking between chunks and commits is based on checksum matching. After splitting into chunks, we filter out the unchanged code lines of a submitted chunk, remove all white space and capitalization, then concatenate all lines into one line. After prepending the relative path name of the file changed by the chunk, we calculate the MD5 checksum. We perform the white space, capitalization and concatenation to deal with small changes done to a chunk before merging, and the path name prepending to avoid false positive matches with similar changes to other files. We perform the same procedure to the commits in the Git repository (after splitting them into chunks), then match the MD5 checksums of submitted chunks and Git chunks. We only link to the closest Git chunk in time that follows the submitted chunk. Overall, 47% of the chunks in the mailing lists could be mapped to Git commits.

[1] A cryptographic algorithm used to generate a 128-bit hash value (checksum), which is often used to verify and validate the data being transformed.

We found that patch chunks, i.e., all changes in a patch to one particular file, are a good compromise between granularity and patch identity. We evaluated the recall of the chunk linking mechanism on a sample of 3,000 email threads from the linux-tips mailing list, which contains the actual Git commit identifier for an accepted patch. Our linking approach had a recall of around 75%. We then performed a random sampling of 100 emails (50% linked and 50% not linked by our linking mechanism) across all mailing lists, to obtain a confidence interval for the mechanism's precision, with length 10% and a 95% confidence level. We found that our technique had a precision of 100%±10%. These numbers provide us confidence that our linking mechanism is sufficiently accurate.

Finally, to map back from the chunk level to the patch level, we analyzed each patch's chunks and considered a patch to be accepted if at least one of its chunks was mapped to a Git commit. The patch's reviewing, integration and staging time are the corresponding times for the first chunk that was accepted by Linus Torvalds. This makes sense, since one accepted chunk is enough for a developer to know that his work will appear (at least partially) in the next release. Of the 348,184 Git commits in Linus Torvalds' repository, 256,284 were actual patch submissions (i.e., non-merge commits authored by other people than the committer), and we could map 190,931 of those commits (74.5%) to a patch submission email.

4.3.3 Measuring Patch Characteristics

In order to study the relation between patch characteristics and (time to) acceptance, we need to define a list of relevant characteristics. Initially, we based ourselves on a list of guidelines published by the Linux Foundation to support kernel developers in getting their patches accepted [65]. Some of these guidelines, such as whether or not a developer has used the "checkpatch.pl" tool before submitting her patch to a mailing list, could not be measured easily. Others, such as whether the patch is small enough or was sent to the right mailing list, were straightforward to measure. We extended this set of metrics with additional metrics, such as whether a patch fixes a bug or proposes a new feature.

Table 4.1 shows all metrics that we used during our study, including the type of metric, the data source from which we calculated it and a short description. They are grouped into five dimensions, and an additional group of dependent metrics. Most of the metrics are straightforward to understand, hence we only discuss some of them in more detail. We consider a review to be each email in a thread that replies to an email with a patch, but only

[1]"Recall" indicates that out of all existing correct items, how many are selected in experiment. Recall is normally used to measure the result integrity.

[1]"Precision" indicates that out of all selected items, how many of them are correct. Precision is normally used to measure the result accuracy.

Table 4.1 Overview of the metrics and dimensions used. The superscript after the name is the research question in which it was used.

	metric name	type	source	explanation
Experience	msg_exp[2,3]	numeric	mbox	Number of patches sent by author in earlier threads.
	commit_exp[2,3]	numeric	git	Number of accepted Git commits thus far by the author.
Email	year/month/week/day[1,3]	nominal	email	Year/Month/Week of the year/Day of the week on which patch was sent.
	nr_ccs[2,3]	numeric	email	Number of people CCed by email.
	msg_length[2,3]	numeric	email	Number of lines of email text excluding patch lines.
	rel_quarter[2,3]	nominal	email	Quarter of release window in which patch is submitted.
	thr_first[2,3]	boolean	thread	Is this email the first one of the current thread?
	first_patch[2,3]	boolean	thread	Is this first patch in thread?
	thr_volume[2,3]	numeric	thread	Number of email messages between start of current thread and current patch.
	thr_part[2,3]	numeric	thread	Number of people participating in current thread until current patch.
	thr_time[2,3]	numeric	thread	Discussion time (seconds) between start of current thread and patch.
	right_venue[2,3]	boolean	patch	Is the patch sent to the most specific mailing list for the changed subsystem?
	lkml_first[2,3]	boolean	thread	Was the first email sent to the general-purpose LKML list?
Review	nr_reviewers[2,3]	numeric	thread	Number of different people sending review messages.
	nr_reviews[2,3]	numeric	thread	Number of review messages.
	response_time[2,3]	numeric	thread	Time in seconds from patch to first review message.
	first_response_time[2,3]	thread	patch	Time in seconds from first patch in thread to first review message.
Patch	bug_fix[2,3]	boolean	patch	Is patch a bug fix?
	chunks_in[3]	numeric	git	#chunks in patch accepted by Linus Torvalds.
	chunks_out[3]	numeric	git	#chunks in patch rejected by Linus Torvalds.
	size[2,3]	numeric	patch	Patch churn (sum of added and removed lines).
	spread[2,3]	numeric	patch	Number of files changed by patch.
	spread_subsys[3]	numeric	patch	Number of subsystems changed by patch.
	nth_try[2,3]	numeric	thread	What version of this patch are we (relative to this thread)?
	commit_sub[2,3]	numeric	git	Number of previously accepted patches modifying the changed subsystem (prior churn).
	patch_set[2,3]	boolean	email	Is this patch part of a larger patch set?
Commit	committer[3]	numeric	git	Number of different committers for the chunks of this patch.
	cc_is_rev1[3]	boolean	git	Is name of committer same as name of first CC-ed reviewer?
	cc_is_rev[3]	numeric	git	Number of chunks where the name of committer is same as name of first CC-ed reviewer.
Dependent	**accepted**[1,2]	boolean	git	Was this patch accepted by Linus Torvalds?
	reviewing_time[1,3]	numeric	email/git	Time between patch submission and first commit by a kernel maintainer.
	integration_time[1,3]	numeric	git	Time between first commit by a kernel maintainer to acceptance by Linus Torvalds.
	staging_time[1,3]	numeric	git	Time between acceptance by Linus Torvalds and the next release.
	total_time[1,3]	numeric	email/git	Time between patch submission and the release in which the patch appeared.
	missed[2]	numeric	email	Number of missed kernel releases since patch submission for an accepted patch.

until the next patch in that thread or the end of the thread.

Developers sometimes want one or more specific persons to review their patches, often the maintainer of a subsystem. To (try to) achieve this, they add those persons' email addresses as the CC addressees of the email (in order of importance). We measure the number of CC-ed people, and also check whether the first CC-ed reviewer indeed committed chunks of the patch.

Since the time in between Linux releases varies between 2 and 3 months, the merge window can also vary in time. As Linus Torvalds determines by himself when the merge window ends, we could not identify the exact dates. Instead, we divided the period in between two subsequent releases into 4 periods ("quarters"), and report for each patch the `rel_quarter` in which the patch was submitted to a mailing list for review.

To identify the most specific mailing list for a particular patch, we analyzed all patches of each mailing list, and counted the number of patches modifying each subsystem. We assigned the most changed subsystem to each mailing list. A subsystem here corresponds to a subdirectory in the kernel code base, such as "kernel" and "net/ipv4". We did not go deeper than 2 levels of subdirectories, since otherwise the subdirectories became too specific.

To tag a patch as being a bug fix, we used a traditional, simple heuristic: We searched in the patch's log message for the words "bug" and/or "fix" (case-insensitive). More advanced techniques exist [152], and we plan to experiment with these in future work.

Furthermore, to identify if a patch is part of a patch set, we used the naming convention suggested by the Linux Foundation [65]. Messages that are part of a patch set should contain the term "PATCH" (case-insensitive) and a marker of the form "5/20", which means that this email is patch 5 out of 20. We check all subsequent threads of the same author within one hour to see if other messages contain "PATCH" and "20". Using a 1-hour window for each email suffices since manual browsing of patches showed that patch set emails follow each other very closely. We cannot check for threads with exactly the same subject, since different patches in a patch set typically have specific subjects for that patch. If all patches of a patch set end up in the same thread, we consider the reviews in the thread to be shared by all patches in the set.

Finally, we only calculate the reviewing, integration and staging time for fully accepted patches. Patches that failed reviewing or went through reviewing, but not through integration, do not have any of these three metrics. For the patches that made it to Linus Torvalds, we measured the shortest time from the initial maintainer commit until a merge commit performed by Torvalds that brings the patch into his repository.

4.3.4 Data Analysis

To address the three research questions, we performed empirical analysis on the metrics that we collected, and also built decision tree models. A decision tree is a tree where every node is a condition such as "size>50?" and one takes a different path based on the evaluation of the condition for a particular patch. The leaves of the tree correspond to a classification such as "accept" or "reject". We provide more explanation when discussing the individual research questions.

In order to improve the performance of the models and to make the values easier to interpret, we discretize the reviewing and integration time variables. For example, it does not really make a difference whether a patch will take 13 or 14 days to review, but it does make a huge difference if a patch will take a month instead of a day. After analyzing histograms of the data, we decided to discretize the time data into the following bins: "instantly", "within_hour", "within_day", "within_week", "within_month", "within_quarter", "within_half_year", "within_year" and "took_ages".

4.4 Case Study Results

This section presents the results of our three research questions. For each question, we present its motivation, the analysis approach and a discussion of our findings.

RQ1: What percentage of submitted patches has been integrated successfully, and how much time did it take?

Motivation: Existing reports about Linux kernel development show that in between 8,000 and 12,000 patches are accepted into each current kernel release, authored by more than 1,000 developers [66]. Although this illustrates the huge scale of development for the Linux kernel, not much is known about the success rate of patches submitted to the kernel. How many patches are actually submitted, and how many eventually make it? Of those who make it, how much time do they need until they are integrated into the next kernel releases? How much of that time is spent on reviewing compared to integration effort? The answers to these questions provide insight into the process, output, productivity and effectiveness of collaborative open source development at massive scale in the Linux project.

Approach: We compute the number of accepted and rejected patches that were sent to a kernel mailing list between 2005 and 2012 and measure the total time it took for the accepted patches to end up in a release, as well as the reviewing, integration and staging time.

Findings: **The yearly number of submitted patches to kernel mailing lists keeps on increasing.** Figure 4.2 shows that, starting in 2005, the number of submitted patches discussed on the mailing list has kept on increasing, with a temporary slowdown from 2008 to 2010. The numbers are staggering: In 2013 alone 136,932 patches were submitted to Linux mailing lists. Even though the number of submitted patches has increased, the percentage of accepted patches has remained between 27 and 34% (see Figure 4.2). The rest (more than 66%) have not made it, but it is hard to know why they were refused. This might be due to some patches being reworked and resubmitted, others still being discussed for consideration, and some might simply be ignored and never applied. Therefore, it is important to know what is the expected time it takes to have a patch incorporated into the kernel.

Patches take 3-to-6 months towards inclusion in a kernel release. With respect to the total time it takes a patch to be released as part of the kernel, Figure 4.3 tells us that in 2005, most of the accepted patches took between 1-to-3 months. Afterwards, the time most patches took grew to 3-to-6 months, remaining relatively stable from 2007–2012. To better understand this, we have decomposed the total time into reviewing, integration and staging time, and analyzed each of them separately.

The distribution of reviewing time has become shorter (Figure 4.4a), with more patches being reviewed within a day and a shift from 1-to-3 months to within 1-week-to-1-months. At the same time, less patches take longer than one month to be reviewed. A non-parametric Kruskal-Wallis test, followed by pairwise Mann-Whitney tests with a confidence level of 0.05 (using Bonferroni correction) confirmed the changes in reviewing time. This suggests that Linux has improved its reviewing process over time. Either reviewers became more efficient, or the patches became easier to review.

On the other hand, the time of integration has slowed down. Figure 4.4b shows that in 2005 almost 60% of patches were committed directly ("instantly") to the Linux kernel by Linus Torvalds and almost 80% made it within a day, while in 2012 only about 10% were integrated within a day. There is a clear increase of the percentage of patches taking a month or even a quarter to be integrated. Again, a non-parametric Kruskal-Wallis test, followed by pairwise Mann-Whitney tests with a confidence level of 0.05 (using Bonferroni correction) confirmed these changes in integration time.

These observations can only be explained by Linus Torvalds applying less and less patches over time. We tested this assumption by computing the proportion of commits authored by other developers but directly committed by Linus Torvalds relative to all non-merge commits. Starting in 2005, he has committed yearly 39%, 30%, 21%, 11%, 7%, 5%, 4% and 4% of all non-merge commits. In 2005, 4,508 patches were accepted for which Linus made

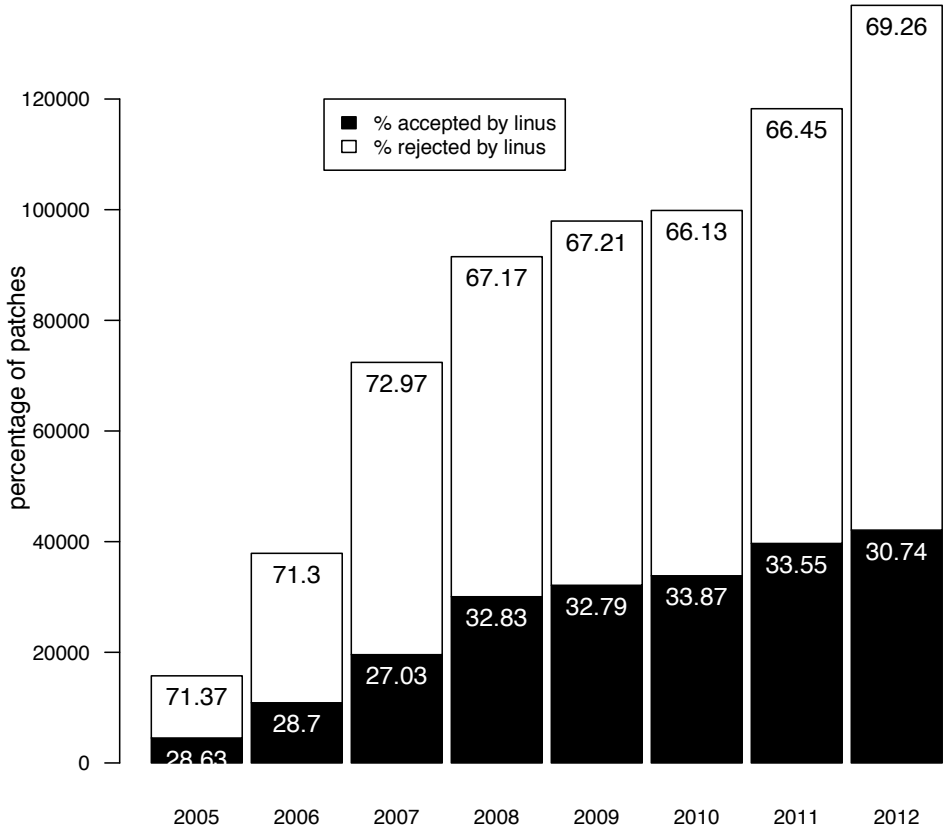


Figure 4.2 Number of accepted patches. The numbers in the bars correspond to the percentage of accepted or rejected patches.

5,623 commits authored by others, while in 2012, there were 42,088 patches accepted and Linus committed others' work only 2,370 times. Similarly, the number of other committers who commit changes authored by another person has grown from 64 to 261. This suggests that Linus is integrating patches less (in addition to developing less himself), delegating this task to others in order to cope with the overall growth in patches. Similar observations were made elsewhere [66].

To understand whether the speed-up of reviewing time and slow-down of integration time is due to changes in development style, we analyzed characteristics of the patches that were reviewed and/or integrated. We found that the size of patches has changed, dropping initially from a median size of 61 lines of code in 2005 to 29 lines in 2007, then increasing again up to a median of 77 in 2012 (87 in 2011). These larger patches are affecting more files as well, since the median number of changed files increased from 3 to 4. Finally, the percentage of bug fix patches has dropped continuously from 24% to 16%, i.e., more patches contain feature enhancements instead of bug fixes. This provides an explanation of the growing size of patches as well as the longer time it takes to integrate these more complex patches. It is not clear why reviewing is not affected by this.

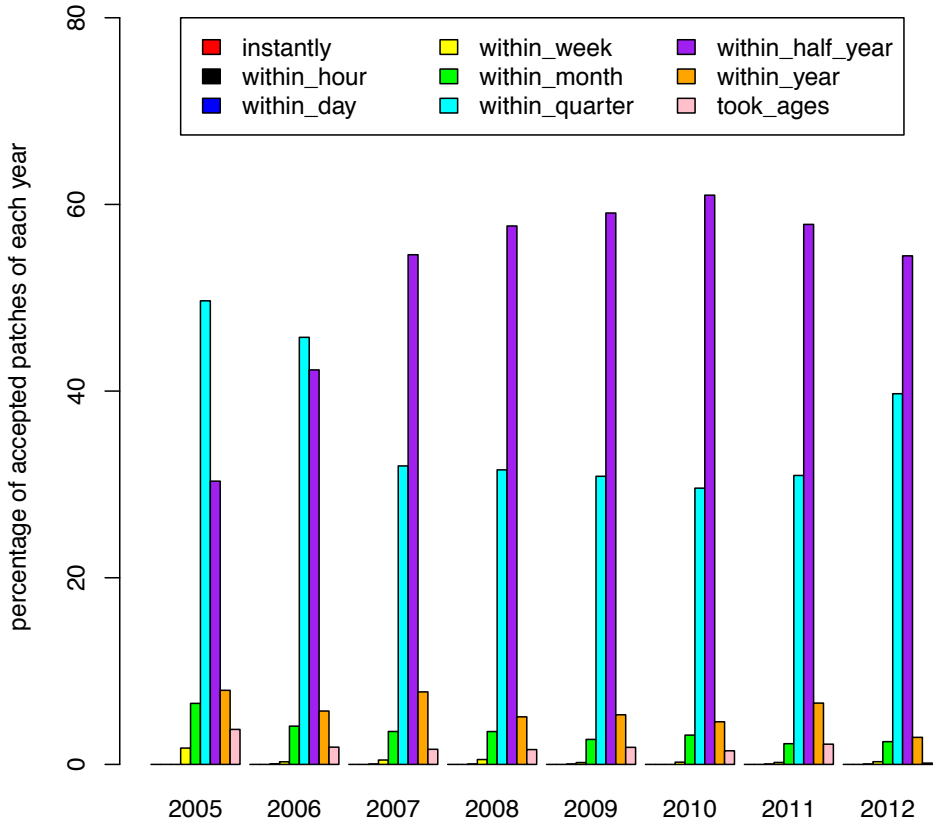


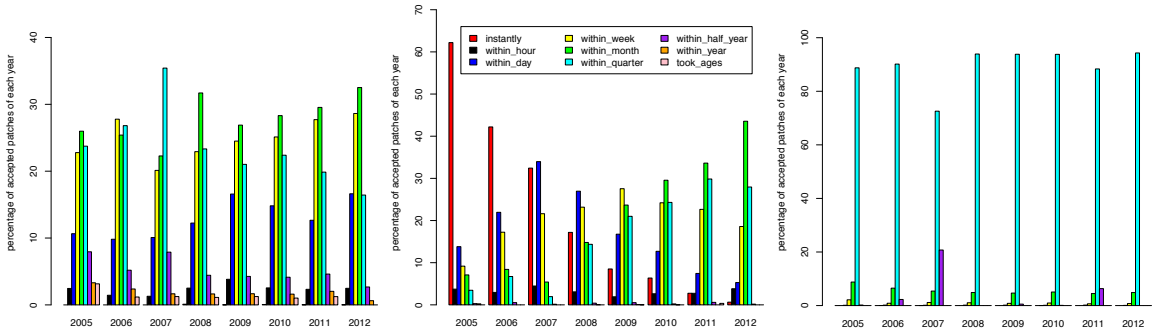
Figure 4.3 Total time for a patch to occur in an official release.

Staging time is fixed at 1–3 months. Figure 4.4c shows how most of the patches got in within one quarter. Indeed, the time in between releases has been set to a fixed number of weeks by Linus Torvalds, i.e., it is a management decision. In addition, patches are only merged into Linus Torvalds' repository roughly the first two weeks after the previous releases. Commits sent outside this merge window are ignored until the next release's merge window. Only in 2007, many patches took more than half a year to be accepted. Since staging time is dictated, we do not consider it in this paper.

The reviewing time is becoming shorter and shorter, seemingly at the expense of a longer integration time. However, the total acceptance time does not change too much.

RQ2: What kind of patch is accepted more likely?

Motivation: In RQ1, we found that the accepted patches represent only around 30% of all submitted patches. Furthermore, it takes several months before knowing that a patch has passed review, and even more time before knowing that it is accepted by Linus Torvalds. Hence, in the worst case, developers can lose a lot of time working on and maintaining a patch that will never make it. Even worse, they could find themselves having to choose between



(a) Reviewing time (b) Integration time (c) Staging time

Figure 4.4 Distributions of times for patches

different bugs to fix or features to implement. The same problem applies to maintainers and integrators, who are buried under a large load of incoming patches and need to separate the wheat from the chaff. Thus, RQ2 examines the characteristics of successful patches.

Approach: As we saw in RQ2 that Linux has a release every 2 or 3 months [66], i.e., roughly once every quarter, we build decision trees for every 3 months of development. In order to understand the characteristics of accepted patches, we build decision tree models with the metrics in Table 4.1 as independent variables and `accepted` as dependent variable. This results into 32 decision tree models from 2005 to 2012.

For each model, we use 10-fold cross-validation to obtain more accurate performance measures. The data basically is split into 10 folds, and we use each fold once as test set with the other folds as training set. This generates 10 trees for each quarter. We use precision (how many patches classified as "accepted" really were "accepted"?) and recall (of all "accepted" patches, how many did we classify as "accepted"?) to measure the performance of the decision trees. As baseline to compare the performance to, we use a zeroR model, i.e., a model that always predicts "accepted". For example, if there are 13% accepted patches, then a zeroR model would have a prediction and recall of 13%.

Then, we analyze the most influential metrics in the models using top node analysis, i.e., we look up the metrics that occur in the conditions of the top 2 levels of each decision tree, since they contain the most decisive information regarding patch acceptance. To visualize the results of top node analysis across all quarters, we use heat maps. These are plots with time as the X axis and the metrics as the Y axis. For a particular quarter and metric a cell is non-white if that metric was a top node in at least one of the 10 trees of that quarter. The darker the cell (light grey up to black), the more trees contained the metric in their top 2 levels.

Findings: **We are able to build high-performing models with up to 73% precision**

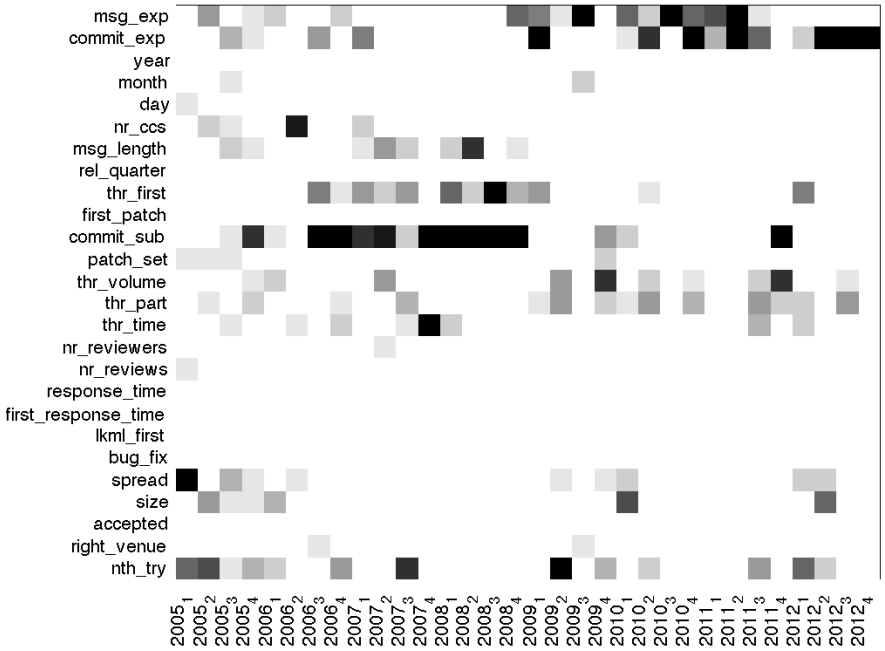


Figure 4.5 Top node analysis results for the patch acceptance models (RQ2).

Table 4.2 Significant attributes for patch acceptance.

Attribute	Influence
`msg_exp`	Smaller value leads to acceptance.
`commit_exp`	Larger value leads to acceptance.
`thr_part`	Larger value lead to acceptance.
`nth_try`	Larger value leads to acceptance.
`commit_sub`	Smaller value leads to acceptance.

and recall. The precision values of our models range from 52% to 73% with an average of 66.28%, compared to 40.41% for the zeroR models (between 17% and 48%). Similarly, for recall we see that the recall values of our models range from 63% to 73% with an average of 68.47%, compared to 40.41% for the zeroR models.

Developer experience, patch maturity and prior subsystem churn are the major factors impacting acceptance of a patch. Figure 4.5 shows the top node analysis results of the decision trees for patch acceptance, while Table 4.2 lists the 5 most impacting metrics overall with their interpretation.

The top two metrics (`msg_exp` and `commit_exp`) relate to developer experience. Developers who have had a commit being merged into Linus Torvalds' repository before have a higher chance of getting their patch accepted, presumably since they have more experience with the creation and development process of kernel patches, and maybe a higher standing in

the community. Surprisingly, having posted more emails (and hence reviews) to mailing lists before has a negative impact on acceptance. It is not clear why this is the case. We hypothesize that those experts might be working on more complex features, which are more risky and prone to rejection. However, more work is needed to verify this hypothesis.

The third and fourth most important metrics are related to patch maturity, i.e., the amount of reviewing discussion preceding a patch submission (`thr_part`) and the number of previous iterations of the patch in the same thread (`nth_try`). Unsurprisingly, the more mature a patch, the higher the probability of acceptance.

The fifth most important metric relates to the number of previous commits to the changed subsystem, i.e., the subsystem's prior churn (`commit_sub`). According to the data, the more popular a subsystem is in the Git commits, the lower the probability that a new patch will make it. It is not clear what exactly this implies. It might refer to the fact that subsystems that see lots of change are harder to keep up with, causing new patches to be outdated by the time they are proposed. Alternatively, it could mean that there is a lot of duplicate work going on, with multiple developers trying to compete for the same features. More analysis is needed to fully understand the impact of this observation.

Among all the above attributes, only the top 2 experience attributes can be controlled by developers by participating more actively in the kernel community. This helps them learn more about kernel development, and to earn the trust of other developers and maintainers. The other 3 attributes are hard to manipulate for a developer.

The major metrics vary over time. We can see how no metric is important in all 8 studied years. Instead, there seem to be roughly three phases in Figure 4.5, i.e., 2005/2006, 2007/2008 and 2009–2012. Phases one and three especially value metrics on patch maturity, patch characteristics, review activity and experience, whereas phase two especially focuses on review activity. We suspect the change in phase two at the end of 2008 to coincide with the introduction in February 2008 of the linux-next integration repository. This repository was introduced to test how a patch would merge with the current version of the Linux kernel, in order to avoid that patches only are merged six months later by Linus and fail at that time. As such, linux-next aimed to increase acceptance rate and reduce integration time through faster feedback.

Patch characteristics, time of submission and review response time do not play a major role in patch acceptance. Looking at the white areas in Figure 4.5, we can see that patch characteristics like size, spread and whether a patch is sent in one piece or has been split into smaller parts (patch set) only played a role initially. This contradicts existing work [136, 156], as well as the Linux Foundation's guidelines [65]. The time in a

release quarter (`rel_quarter`) does not play a role either, and neither does the quantity and responsiveness of reviews.

The most important metrics for patch acceptance are changing all the time. We find that the patches of experienced developers, mature patches and patches in subsystems with less prior churn are easier to be accepted.

RQ3: What kind of patch is accepted faster?

Motivation: Similar to the motivation for RQ2, we find that some patches took much longer than others to arrive to Linus Torvalds' repository. For developers, maintainers and integrators, it is important to understand which patch properties determine reviewing and integration time. Hence, we will analyze only patches that eventually were merged by Linus Torvalds.

Approach: Similar to RQ2, we build decision tree models for each quarter. This time the dependent variable is either the discretized reviewing time or integration time. We again use 10-fold cross-validation for the reviewing time and integration time models, followed by top node analysis. Instead of precision and recall for each of the 9 outcomes, we compute the global accuracy, i.e., the percentage of correctly classified patches (across the 9 outcomes) relative to all patches.

Findings: **The reviewing and integration time models obtain an accuracy of up to 70% and 76%, resp., and again evolve over time.** The accuracy values for the reviewing time model fluctuate between 53% and 70% (mean of 59%), while for the integration time they fluctuate between 62% and 76% (mean of 68.4%). Although the models evolve over time, we observe more noise in the top node heat maps (Figure 4.6 and Figure 4.7) compared to the acceptance models. It is hard to spot different phases, except for the reviewing time models, where the experience-based metrics at the top stop being influential. It is unclear why this is the case (linux-next only applies to integration, not to reviewing).

For reviewing time, a variety of dimensions is influential. Figure 4.6 and Table 4.3 show the following top 6 influential dimensions: Prior subsystem churn, time, patch characteristics, reviewers and experience. Contrary to the model in RQ2, most of these attributes (except the prior subsystem churn) can be directly influenced by a developer. Prior subsystem churn plays a rather complicated role, changing its impact across different quarters. Since bugs need to be fixed irrespective of a subsystem, one cannot really tweak this attribute. On the contrary, the week of the year in which one submits a patch for review plays a significant role. This is tied to the merge window of a release, i.e., the period in which

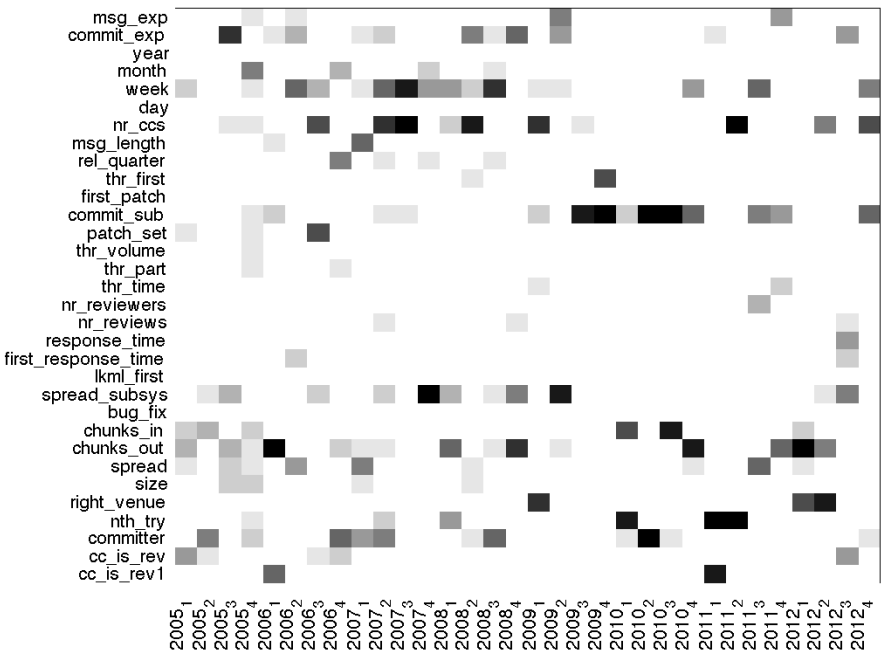


Figure 4.6 Top node analysis results for reviewing time.

the new submissions are solicited. If timed properly, a submission of a patch will arrive in time for a merge window, otherwise it might be delayed at least an entire cycle until the next release merge.

Various patch characteristics, such as the `spread` of a patch and the number of chunks in it play an important role. In fact `chunks_out` represents the number of chunks that were not committed in the end, i.e., the chunks that were ruled out during reviewing or integration. Deciding about this indeed takes up quite some reviewing time. This brings us to the reviewing dimension (`committer` and `nr_ccs`). `committer` corresponds to the number of different people committing (and hence probably involved in reviewing) chunks of a patch, whereas `nr_ccs` refers to the number of people suggested as reviewer by putting them into the list of cc-ed email addresses. Hence, the choice of reviewers indeed plays an important role in reviewing time [48]. Finally, experience again plays a major role.

Surprisingly, some metrics that do not play a role are whether a patch fixed a bug (we expected bugs to be expedited), nor whether the patch had been posted to a specific mailing list for a subsystem vs. the global LKML. This finding contradicts the Linux Foundation guidelines [65].

For integration time, the same dimensions are influential as for probability of acceptance. In Table 4.4, the only difference is that a high `msg_exp` this time leads to shorter integration, whereas it decreased the probability of acceptance (RQ2). Furthermore,

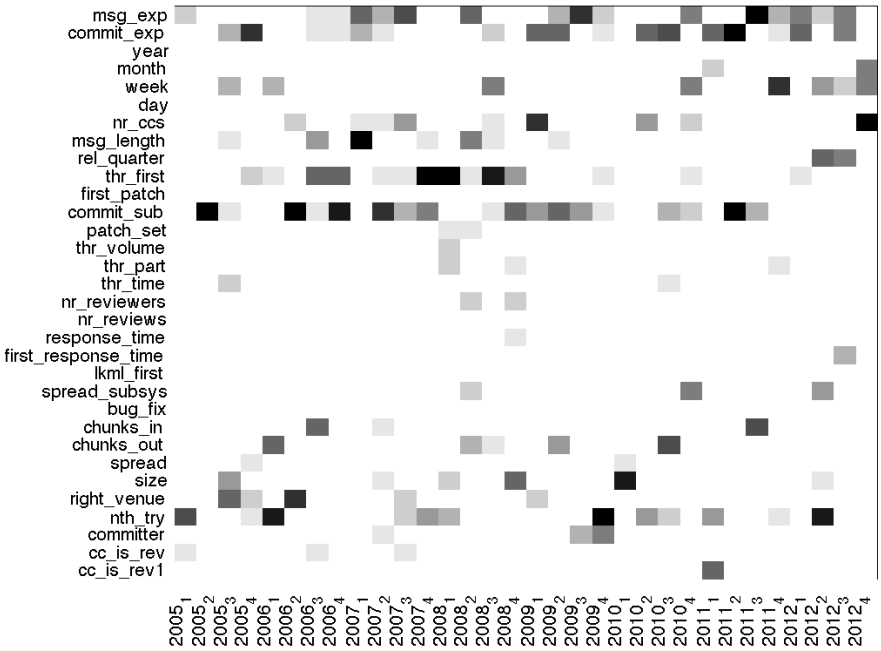


Figure 4.7 Top node analysis results for integration time.

experience plays a much larger role for integration than for reviewing. The fact that the models for acceptance and integration value the same attributes suggests that integration plays an essential role in patch acceptance, maybe more than reviewing by itself.

Developers can reduce reviewing time by controlling the submission time, the number of affected subsystems by the patch, the number of reviewers CCed on the mailing list submission, and by being more active in the community. Gaining experience through active participation in the open source community is also linked to shorter integration time.

4.5 Threats to Validity

We now discuss the threats to validity of our study, following common guidelines for empirical studies [162].

Construct validity threats concern the relation between theory and observation. We use decision trees to find important relations between a rich set of metrics and patch acceptance (time). Since these models are not perfect (precision, recall and accuracy below 100%) and are statistical in nature, they might not correspond to actual relations or actions in the development process. Although qualitative analyses are needed to validate these models, we built many models across time and observed various recurring phenomena, giving us confidence that the models do provide major indications about the relations. Furthermore, our selection

Table 4.3 Significant attributes for reviewing time.

Attribute	Influence
commit_sub	Impact varies a lot across time, but in recent years larger values lead to shorter time.
week	Larger value leads to longer reviewing time.
chunks_out	Larger value leads to longer reviewing time.
spread_subsys	Larger value leads to longer reviewing time.
committer	As the reviewing time increases, the number of committers first increases, then decreases.
nr_ccs	Larger value leads to longer reviewing time.
commit_exp	Larger value leads to shorter reviewing time.

Table 4.4 Significant attributes for integration time.

Attribute	Influence
msg_exp	Larger value leads to shorter integration time.
commit_exp	Larger value leads to shorter integration time.
thr_first	"true" value leads to shorter integration time.
nth_try	In early years, having more iterations leads to shorter integration time, but recently the opposite holds.
commit_sub	Larger value leads to shorter integration time.

of metrics is based on guidelines provided by the Linux Foundation [65], enhanced with other patch-related metrics.

Threats to internal validity concern our selection of subject systems, tools, and analysis method. The automatic mapping between email patches and Git commits contains many different steps that might introduce noise. First, the mailing list archives typically contain incomplete information (such as emails that are not clearly linked to a thread, which even specialized tools cannot resolve perfectly [54]. Second, the chunk-based linking approach cannot handle changes to identifier names (contrary to Bird et al. [57]). Also, even though the chunk-level is sufficiently coarse-grained, it is still possible that similar changes are made to the same file in a short timespan, for example on different clones in a file. However, our linking approach obtained a high precision and recall when evaluated on a sample of the data. Third, Git does not retain information about the name of the repository or branch where a commit originated, and it even allows developers to change (clean up) the history of changes in order to make later integration easier [59]. This might affect the heuristics and other techniques used to calculate the integration time of a commit.

Threats to external validity concern the possibility to generalize our results. Since we have only studied one large open source system, we cannot generalize our findings to other open

and closed source projects, even in the same domain. Furthermore, since Git is a flexible version control system that can be used in various setups, we also need to take care when extrapolating our findings to other projects using Git. Hence, more case studies on other projects are needed.

4.6 Related Work

Our work is related to previous studies on reviewing processes in open source projects, on software integration and on prediction of bug fixing time.

Rigby et al. [136] studied reviewing practices in the Apache open source system, and compared these to those of a commercial system. Between January 1997 and October 2005, the Apache mailing list contained 9,216 review email messages for 2,603 patches. They found that small, independent, complete patches are most successful, whereas we found that size only plays a role for reviewing time, whereas patch sets do not play a major role. 50% of the patches are reviewed in less than 19 hours, whereas for Linux this takes 1–3 months. Finally, Rigby et al. found that 44% of the submitted patches eventually were accepted, compared to 33% in our study.

Weissgerber et al. [156] studied contributions in two small open source systems (196 and 1,628 patches respectively, changing 6% of all files in the systems), and also found that around 40% of the contributions are accepted. Similar to Rigby et al., they find that smaller patches have a higher probability of being accepted. Most of the patches are accepted in the repository within one week (61% even within three days). Since they looked at CVS repositories, a commit automatically corresponds to a chunk (one file change). These chunks were mapped to mailing list patches by looking for the changed lines in the files (irrespective of their order), whereas we concatenated all changed lines and removed all whitespace and capitalization to improve performance. One quarter of the accepted patches took less than one day for reviewing and integration, half were accepted within one week, and one third took longer than two weeks. No relation between patch size and the time towards acceptance could be found. Similarly, we only found a link between patch size and reviewing time, but not with integration time.

Baysal et al. [48] studied reviewing effort in the Mozilla project. Contrary to the previous two studies and our study, Mozilla uses a modern web-based system (Bugzilla) to manage review requests, comments and their outcome. On average, 78% of the reviewed patches made it through reviewing, but only around 60% of those (i.e., 47% in total) eventually make it into the code repository, which is slightly higher than the other studies. Similar to

us, ore contributors have a higher probability of getting their patches accepted in the code repository, although experience does not impact the outcome of the reviewing phase (it does impact the reviewing time in our case). Similar to Linux, people can suggest reviewers and this choice plays an important role to avoid having a patch end up to be rejected. Mozilla patches that make it into the code base took 4.5 days before the switch to rapid release, and 2.7 now for core developers (slightly less time for casual developers). For the Linux kernel, this can take up 3–6 months.

Our work differs from the work above in multiple ways. First off, we studied the whole integration process, i.e., not just reviewing. Furthermore, we studied the relation between more than thirty metrics and patch acceptance (time) and how this relation evolves over time. Finally, Linux has a much deeper hierarchical structure of contributors and maintainers, whereas projects like Apache and Mozilla are relatively flat. As a consequence, the time to get through reviewing or merging is much shorter than is the case for Linux, and the impact of merging becomes more important.

Regarding software integration, various researchers have studied the impact of branch structure (e.g., the hierarchical structure of source code repositories in Linux) on software quality. Bird et al. [60] proposed a what-if analysis that allows to identify harmful, redundant branches in order to simplify the branching structure. Such simplification can avoid merge conflicts (incompatible changes in the two versions of the code base), which inflate integration time. In their case study on a large commercial system, such a simplification was able to reduce integration time by up to 9 days. Shihab et al. [146] found how a mismatch between the branching structure and a company's organizational structure can increase post-release failure rate. This is why Linux kernel development uses a distributed version control system.

Brun et al. [64] studied different kinds of merging conflicts, and identified four major reasons why integration can be risky: (1) The integrator did not develop the code herself, (2) the patches to merge often contain too many changes in one big lump, (3) merging typically happens a relatively long time after the actual development, and (4) most of the serious conflicts are due to semantical instead of textual issues. We indeed found evidence for these four reasons in Linux.

Our models to explain the reviewing and integration time are related to the work on bug fix time prediction, where, based on a bug report, the time for fixing this bug report is predicted. Guo et al. [84] built models to predict which Windows 7 bugs would be fixed. The precision and recall of those models were around 65%. They found that having more people following the status of a bug improves the probability of the bug being fixed. Similarly, having submitted bugs before that ended up being fixed improves the chances of getting a bug fixed

in the future, i.e., experience plays an important role. We made similar observations for our models of acceptance (time).

Giger et al. [80] found that post-submission data of bug reports such as the number of comments or number of interested people improves the accuracy of bug fix time prediction models. However, in our models, adding the number of reviewers and the number of reviews did not make a significant change. Finally, Zhang et al. [164] build a model to predict the time between assignment of a bug report and the time when bug fixing starts. This corresponds to the time between sending an email talking about a new idea for a feature and sending an email with the patch implementing the idea. However, since related email threads are hard to link to each other, we were not able to build models for this incubation time.

4.7 Conclusion

Given the complexity of code integration, and its opaque nature for most developers, we studied the characteristics of patches that could explain patch acceptance and reviewing/integration time. We performed a study on the Linux kernel mailing lists and Git repository, and found that reviewing and integration are two relatively independent processes. Developer experience, patch maturity and prior subsystem churn play a major role in patch acceptance and integration time, while submission time, the number of affected subsystems, the number of contacted reviewers and developer experience correlate the most with reviewing time. The one common thing in the three kinds of models is developer experience, i.e., there seems to be no substitute for active participation in an open source project while learning the project's ins and outs.

Acknowledgments

We thank Amatul Mohosina for her initial work on this topic, and Richard Ellis for kindly providing us access to the Linux kernel mailing list mbox files.

CHAPTER 5 ARTICLE 2: TRACING BACK THE HISTORY OF COMMITS IN LOW-TECH REVIEWING ENVIRONMENTS

Yujuan Jiang, Bram Adams, Foutse Khomh, Daniel German

Abstract

Context: During software maintenance, people typically go back to the original reviews of a patch to understand the actual design rationale and potential risks of the code. Whereas modern web-based reviewing environments like gerrit make this process relatively easy, the low-tech, mailing-list based reviewing environments of many open source systems make linking a commit back to its reviews and earlier versions far from trivial, since (1) a commit has no physical link with any reviewing email, (2) the discussed patches are not always fully identical to the accepted commits and (3) some discussions last across multiple email threads, each of which containing potentially multiple versions of the same patch.
Goal: To support maintainers in reconstructing the reviewing history of kernel patches, and studying (for the first time) the characteristics of the recovered reviewing histories.
Method: This paper performs a comparative empirical study on the Linux kernel mailing lists of 3 email-to-email and email-to-commit linking techniques based on checksums, common patch lines and clone detection.
Results: Around 25% of the patches had an (until now) hidden reviewing history of more than four weeks, and patches with multiple versions typically are larger and have a higher acceptance rate than patches with just one version.
Conclusion: The plus-minus-line-based technique is the best approach for linking patch emails to commits, while it needs to be combined with the checksum-based technique for linking different patch versions.[1]

5.1 Introduction

In November 2003, there was a suspected backdoor attempt to taint the source code of the Linux kernel project [129]. The Linux team noticed that a patch popped up in the CVS copy of the version control system that never appeared in the BitKeeper master copy. This patch pretended to just check for errors, but actually contained a back door that could render a

[1]This paper has been published in the conference Empirical Software Engineering and Measurement in 2014.

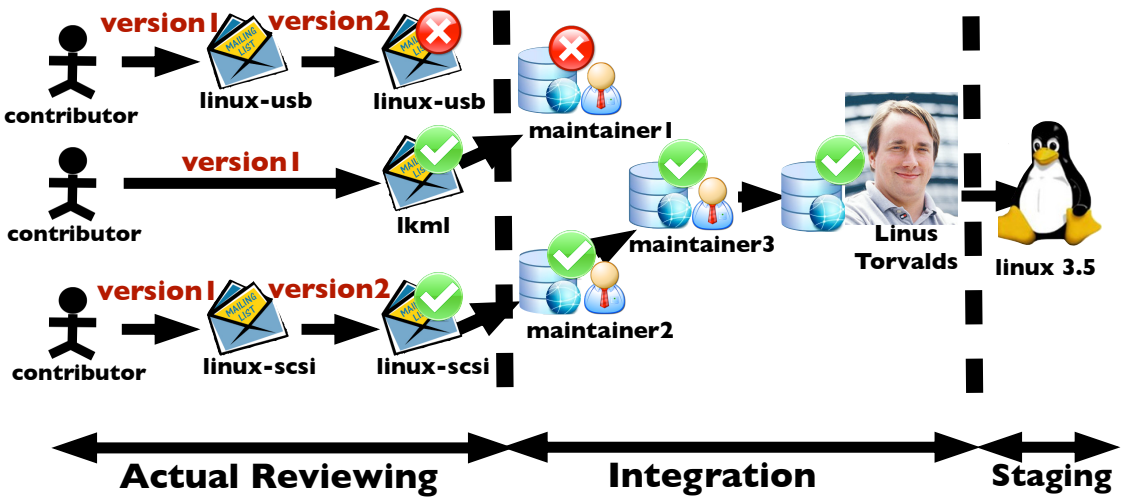


Figure 5.1 The reviewing and integration process of a patch in the Linux kernel project (adapted from [100]).

system vulnerable. Up until today, it is still not clear whether this code was inserted by a malicious hacker, since there was (and still is) no explicit link from a commit back to all emails reviewing that patch or earlier versions of it. The recent "heartbleed" vulnerability [94] also highlights the need to have traceability from commits in the version control system to patches in the mailing list. Full traceability would have made a quicker and more accurate audit of the code possible. Apart from security audits, traceability from submitted patches to accepted commits would also benefit regular maintenance, as it will make it easy to review the email messages around that patch which comprise the reviewing and design discussions of the patches [40][49][136].

Although more and more open and closed source systems are migrating towards modern, online reviewing environments like Gerrit, which consolidate all patch versions, discussions and links to commits in one location, many open source ecosystems such as the Linux kernel and the Apache Software Foundation still swear by low-tech reviewing environments based on a mailing list [51]. Basically (Figure 5.1), a contributor sends an email containing a patch, asking for feedback. Other project members with the required expertise then jump in to provide design- and code-related comments. They either reject the patch up front (not an interesting feature at all), they allow the responsible kernel maintainer to commit the patch to the version control system as is, or they request a new version with modifications. In the latter case, the original patch submitter should send a new patch version, hoping that this one will be successful. This process continues until the patch is rejected, accepted or the submitter gives up.

Although modern reviewing environments require the same steps, the usage of a low-tech

environment introduces three major challenges for people to trace a commit back to the original patch (and its reviews):

- **Reviewing process detached from development process.** Patches and reviews are spread across email messages in one or more mailing lists, while commits are stored in a version control system like Git (see Figure 5.1). Emails cannot reference the commit id of a patch that has not been committed yet, while emails do not have a universal identifier that can be referenced by a commit.
- **Accepted commits can differ significantly from original, submitted patches.** During the reviewing process, the reviewers are likely to ask the developer to revise it. Even after the maintainers integrate the patch into a Git branch, maintainers can still modify a patch by changing the order of commits ("rebasing") or filtering out uninteresting changes from the patch("cherry-picking").
- **Patches evolve across multiple versions, possibly submitted in multiple email threads.** During the reviewing process, a patch may evolve substantially, yielding a series of patch versions. Such a version can be sent in a reply to the original email thread, or sometimes in a new email thread. In the best case, such a sequel thread is announced in the subject of the new thread, for example "[V2]Patch: remove the deadlock code in driver subsystem", however this practice is not enforced and typically only used by more experienced developers.

In this paper, we propose an approach to recover the full reviewing history of a patch in a low-tech reviewing environment. This approach mainly consists of two parts: (1) linking a commit to any email submitting the original patch (often the last patch version), and (2) linking all emails about the same patch together to reach the first version. We used three techniques of different strictness and granularity to link commits to emails and emails to emails: (1) a checksum-based technique (strictest, chunk level), (2) a novel plus-minus-line-based technique (medium strict, line level), and (3) the CCFinder clone detection tool (least strict, token level). After tracing the commits to all emails up to their very first patch version, we are then able to quantitatively analyze (for the first time) the full reviewing history of a patch sent to a mailing list.

We addressed the following three research questions:

RQ1) *Can commits be linked accurately to emails containing the corresponding patch version?*

Since emails and git commits are two different media, we first analyze which technique works best to link an email containing a patch to the accepted commit. The checksum-based technique has the highest precision (97.9%±5%), while the plus-minus-line-based technique has the highest relative recall (85.69%±5%).

RQ2) *Can emails containing different patch versions be linked accurately to each other?*

Since different versions of a patch can be submitted and reviewed across different email threads, we evaluate different techniques for linking related emails to each other. The checksum-based technique again has the highest precision (86.98%±5%), while the plus-minus-line-based technique again obtains the highest relative recall (68.68%±5%). Together, the checksum-based and plus-minus-line-based techniques can discover more than 95%±5% of the correct patch pairs.

RQ3) *What are the characteristics of the reviewing history in a low-tech reviewing environment?*

Given the means to link patch emails to each other and to commits, we can now study the characteristics of patches having multiple versions. We find that around 25% of the patches have a full reviewing history of more than four weeks. Patches that underwent multiple versions have a higher chance to be accepted and especially consist of bug fixes compared to regular patches.

In the rest of the paper, we first introduce the three techniques we applied (Section 5.2) and provide our case study setup (Section 5.3), followed by the findings of our case study (Section 5.4). We conclude with threats to validity (Section 5.5), related work (Section 5.6) and the conclusion (Section 5.7).

5.2 Three Linking Techniques

In general, to recover the evolution process of a commit, we need to (1) trace it back from its final version seen (the accepted commit) to the email submitting the committed version of the patch, then (2) link together all emails containing different versions of the same patch until we reach the first patch version. Since in both phases one tries to map one version of a patch to another one, we use and evaluate the same linking techniques for both. We do not consider the textual content of the reviewing emails themselves, nor commit metadata like the commit log message or author names, since those contain natural language vocabulary, which is known to be much more variable and hence harder to link than source code [92].

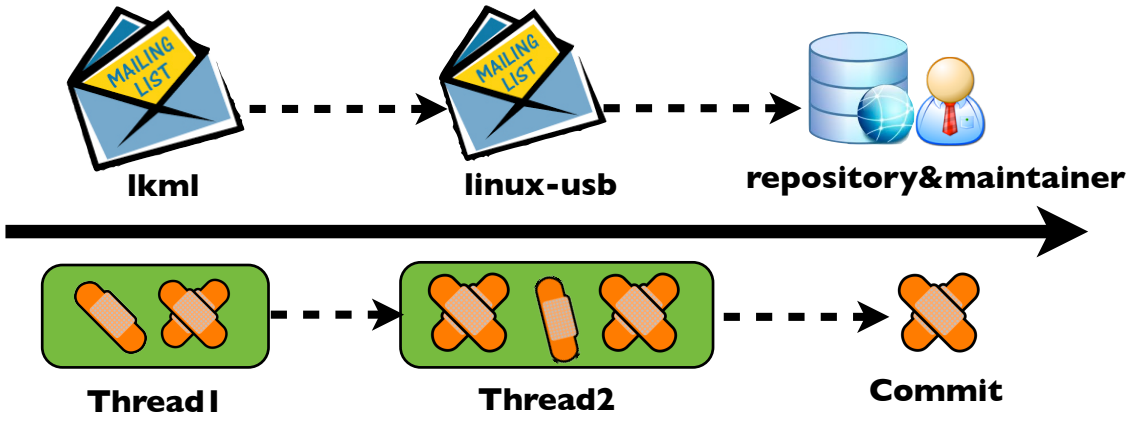


Figure 5.2 The evolution process of a patch with multiple versions ("super-thread").

Figure 5.3 shows an example committed patch in the git format used by Linux and many other open source systems. The part above the "diff −−git" line consists of commit metadata such as the commit id, author name, author data and commit log message, whereas the bottom part contains the actual patch. In this case, the patch describes the changes to a file called "src/networking.c", where on line 989 the developer has removed the line starting with a minus sign and replaced it by the lines starting with a plus sign. The code lines that do not start with a plus or minus sign correspond to the context of the patch, showing the surroundings of the changed lines.

Patches can contain multiple bursts of plus/minus lines (multiple changes to one file), and even multiple "diff −−git" sections changes to multiple files. All changes to one file are called a "chunk". The difference between a committed patch (like Figure 5.3) and a patch version in an email is that for the latter the commit log message occurs as email body, the author name and data correspond to the sender name and date of the email, and there is no commit ID attached to the email, since the patch is not yet committed.

5.2.1 Checksum-based Technique

The checksum-based technique is the strictest and most coarse-grained technique[100]. It computes MD5 hash checksums for each chunk in a commit or email patch, then links each chunk's checksum to the most recent email patch chunk with the same checksum. If an email patch has at least one chunk that was linked to a commit or email chunk, we link the patch as a whole to that commit or email. Finally, we link two email threads together if they have at least one linked email pair.

For each chunk, we first filtered out the unchanged code lines (those without plus or minus sign), remove all white space and capitalization, then concatenate all changed lines (i.e., the

lines starting with a + or - in Figure 5.3) into one line. After prepending the relative path name of the file changed by the chunk, we calculate the chunk's MD5 checksum. We perform the white space, capitalization and concatenation processing to deal with small changes done to a patch before merging, and the path name prepending to reduce false positive matches with similar changes to other files. We perform the same chunk-level procedure to the commits in the Git repository.

Similar to Bird et al. [57], we did not directly link a whole patch to a Git commit, but used the intermediate step of chunks, because of "cherry-picking". This is a common integration activity where an integrator only picks the interesting parts of a patch and ignores the rest. Large patches risk not being merged completely, or maybe not in one Git commit. We link a whole patch to a commit, if at least one of its chunks are linked to that commit.

5.2.2 Plus-Minus-Line-Based Technique

The plus-minus-line-based technique looks for patches containing sufficient identical changed lines (instead of a whole chunk), which is a compromise between both strictness and granularity compared to the other two techniques. Hence, the technique mainly focuses on the lines beginning with a plus or minus sign in Figure 5.3.

The linking algorithm consists of two steps. First, we parse all the Git commits and extract the changed file name, the changed line type (plus + or minus -) and the line content. We save all the information into an sqlite3 database in chronological order. Then, we analyze all emails containing a patch: for each line beginning with a plus "+" or minus "-", we query the database to find other emails or commits containing the same changed line in order to compute the proportion of matched lines with each later patch. Finally, we rank the matched patches according to the proportion of matched changed lines and output the matched patch with the highest proportion. To avoid slight changes to the patch during the reviewing process, we again remove all whitespace of each line (similar to the checksum-based technique).

5.2.3 Clone-Detection-based Technique

As our third technique, we use the popular CCFinderX clone detection, which is a token-based clone detection tool that can detect code clones in C, C++, Java, COBOL, and other source files. It transforms the input source text into tokens, then compares files token by token, and generates pairs of token sequences ("clones") that are found to be very similar (according to some threshold). Since such a technique allows some tokens in the matched

```
commit ff9d66c4a9a6fc91233034bfbcaf5c65379a3bed
Author: antirez <antirez@gmail.com>
Date:   Tue Aug 27 11:54:38 2013 +0200

    Don't over-allocate the sds string for large bulk requests.

    The call to sdsMakeRoomFor() did not accounted for the amount of data
    already present in the query buffer, resulting into over-allocation.

diff --git a/src/networking.c b/src/networking.c
index be78a19..d0d0430 100644
--- a/src/networking.c
+++ b/src/networking.c
@@ -989,13 +989,13 @@ int processMultibulkBuffer(redisClient *c) {
             if (ll >= REDIS_MBULK_BIG_ARG) {
                 /* If we are going to read a large object from network
                  * try to make it likely that it will start at c->querybuf
-                 * boundary so that we can optimized object creation
+                 * boundary so that we can optimize object creation
```

Figure 5.3 A commit patch example. The line with a beginning of a plus sign "+" indicates an added line while a minus "-" sign indicates a deleted line.

token sequences to be different, this is the least strict and most fine-grained technique of the three [55].

We create one file for each patch version and each commit, then feed all files to CCFinderX. Afterwards (for email to commit links), we remove pairs of clones belonging to two emails from the output as well as unfeasible pairs where a commit occurred before the email was sent. Note that to enable CCFinderX to process patches instead of regular source files, we took the changed lines, removed the plus and minus characters, then put braces around the resulting code, forcing CCFinderX to consider the changed lines as a regular code block.

Since CCFinderX is known to produce false positives, we need to filter out (email, commit) or (email, email) pairs with just a few common clones. To help with this, CCFinderX computes some metrics about the analyzed input files and clones, one of which is the RSA metric. This metric indicates the similarity percentage between each file (based on the number of common clones). If two files have an RSA value close to 100%, then one of them is very likely to be a copy of another file. To use the RSA value as a threshold to filter out false positive pairs, we rank all the (email, commit) or (email, email) pairs by their RSA value, see Figure 5.4, then compute the delta in RSA value between each successive pair. We then define as threshold the RSA value of the (email, commit) or (email, email) pair with the largest delta in RSA, i.e., where the slope of the tangential line changes the most in Figure 5.4. We filter out the clone pairs with a value lower than the RSA threshold.

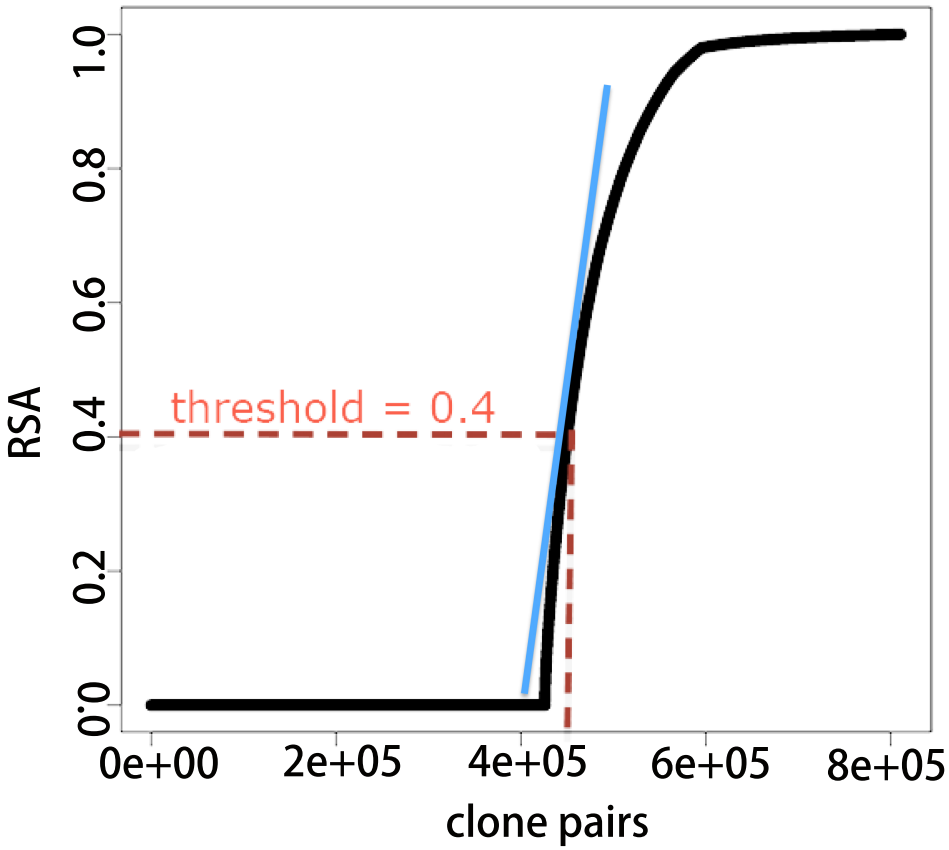


Figure 5.4 The value of RSA where the tangential line (blue) has the largest slope.

5.3 Case Study Setup

In this section, we present the details of the case study that we performed on the Linux kernel data to evaluate the three linking techniques.

5.3.1 Data Extraction

This paper uses the Linux kernel as representative example of an open source project with a low-tech reviewing environment. The Linux kernel reviewing and development process is supported by the Git distributed version control system, the main kernel mailing list (LKML) and more than 130 specialized subsystem mailing lists. These mailing lists are archived online as textual mbox files. After downloading these files from 2009 to 2012, we imported their content and metadata, e.g., subject and author, into a relational database using MailMiner [54]. We ignore email attachments, since the Linux kernel developer guidelines state that all patches should be inlined into the email body.

To analyze those patches that are successfully integrated into an official Linux release, we collect the commits of Linus Torvalds' repository from 2009 (start reviews data) until 2012,

which contains the commit information of accepted patches (Figure 5.3). Note that some emails of end 2012 will not have had the chance of reaching Git, however this does not impact our findings.

5.3.2 Evaluation of Linking Techniques

To evaluate the performance of the three techniques for RQ1 and RQ2, we compute precision and **relative** recall for each, and compute "real" recall for RQ2.

In both cases, we manually analyze precision. We sample 384 pairs from the results of each technique to obtain a confidence level of 95% and confidence interval of 5% [96]. Then, we run scripts to collect the Git log of a commit and the contents of the email to which a commit has been mapped in order to manually check attributes such as the title, author and review comments to determine if the matched commit and patch are really talking about the same patch.

Since pure recall is hard to compute due to the lack of ground truth, we use the concept of **relative recall** in order to measure the sensitivity of the three techniques. Such **relative recall** is defined as the number of correctly detected (email, commit) or (email, email) pairs out of the union of true positive pairs identified during our manual analysis for precision of the three techniques. Although not identical to pure recall, such a relative recall gives an indication of the amount of false negatives in the results.

For example, Figure 5.5 shows how technique A detects 7 candidates, 4 of them being correct, and how technique B detects 6 candidates, three of which are detected correctly. In this case, the union of all correctly detected candidates is 6, 4 of which are detected by A while 3 are detected by B. Then, the relative recall of technique A is 4 out of 6 (67%) while the relative recall of technique B is 3 out of 6 (50%).

In the case of email to email linking, there is a partial source of ground truth, which we also use to obtain a form of real (i.e., non-relative) recall. Basically, some developers reuse the same subject for different email threads, or add a qualifier like "[v2]" or "[V 3]". Using regular expressions, we recovered such links between threads, then randomly sampled 384 of them (confidence level of 95% and confidence interval of 5%) as ground truth for calculating recall.

Finally, we also compare overlap between the results of each technique to figure out if the techniques are complementary or subsume each other. The overlap is computed by counting the number of detected pairs shared with the other techniques. If two techniques are overlapping substantially, it means that they detect almost the same pairs. On the contrary, if

Table 5.1 Overview of the reviewing history metrics and dimensions analyzed.

	metric name	type	source	explanation
Review	thr_volume	numeric	thread	Number of email messages between start of current thread and current patch version.
	nr_reviews	numeric	thread	Number of review messages of a patch version.
	review_time	numeric	patch	Time in seconds from current patch version to its last review message.
	response_time	numeric	thread	Time in seconds from current patch version to its first review message.
	first_response_time	numeric	patch	Time in seconds from first patch version in thread to its first review message.
Patch	size	numeric	patch	Patch churn (sum of number of added and removed lines).
	spread	numeric	patch	Number of files changed by current patch version.
	spread_subsys	numeric	patch	Number of subsystems changed by current patch version.
	bug_fix	boolean	patch	Does this patch version contain a bug fix (as opposed to a new feature or enhancement)?
	accepted	boolean	commit	Has this email patch version been accepted as a commit?

they are not overlapping too much, they are complementary to each other. In this case, one should probably use the union of their results for further analysis to obtain the best precision and recall.

5.3.3 Analysis of the Reviewing Process

As a first application of the linking techniques, we are able to analyze (for the first time) the reviewing history of kernel commits back to the original submitted patch versions (depending on the performance of the linking techniques). For this analysis, we introduce the concept of "super-thread" (Figure 5.2), which connects all patch versions of a commit to each other chronologically. We call the sequence of all versions of a patch a "super-thread", since the email patches could be spread across one or more email threads. For example (Figure 5.6), if there are three emails patches Ea, Eb and Ec in three different threads Ta, Tb and Tc, and we are able to link Ea to Eb, and Eb to Ec, eventually we could link the three threads Ta, Tb and Tc together. Figure 5.6 illustrates the three different kinds of super-threads: SS (single patch version, single thread), MS (multiple patch versions, single thread) and MM (multiple patch versions, multiple threads).

We use the following steps to identify super-threads: (1) mapping each email to its enclosing thread (based on the mbox data), (2) lifting links between emails up to links between threads, (3) filtering out pairs within the same thread, and (4) transitively chaining (thread, thread)-

Figure 5.5 Example of precision and relative recall.

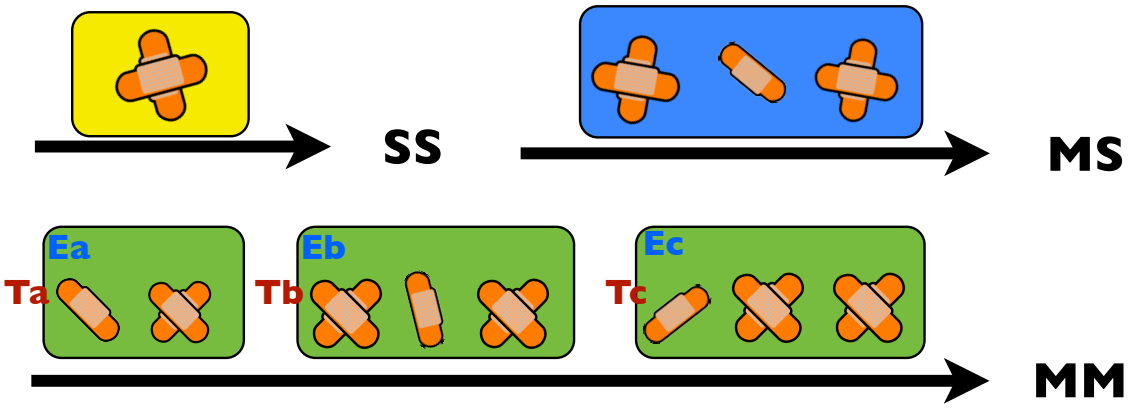


Figure 5.6 The three types of super-threads.

pairs together if they have at least one thread in common.

We use the best of the three linking techniques to recover the (email, commit) and (email, email) links. Especially (relative) recall is important, however we also want to avoid too many false positives. For this reason, we use one or two techniques that together cover a large enough part of the relative recall, but we also take additional measures to keep precision high enough. We will discuss these for RQ3.

After recovering the super-threads, we performed quantitative analysis to analyze the reviewing history and compare important review characteristics. The characteristics that we analyzed are listed in Table 5.1. These characteristics cover two dimensions, i.e., "Review" and "Patch". "thr_volume" measures the reviewing activity for a patch. A higher value means that the patch is being discussed more, "nr_reviews" indicates the number of reviews on a patch version. "review_time" depicts the discussion time for a patch version. The

"response_time" and "first_response_time" metrics measure the reaction of reviewers to a patch. "size" indicates how large a patch is. "spread" and "spread_subsys" metrics depict the invasiveness of a patch. Finally "bug_fix" and "accepted" indicate whether a patch version contains a bug fix (i.e., contains the words "bug", "fix" or "error" [104] in the commit message) and whether it eventually was accepted as a commit.

5.4 Case Study Results

In this section, we answer each research question and present its motivation, approach and findings.

RQ1: Can commits be linked accurately to emails containing the corresponding patch version?

Motivation: In a project like the Linux kernel, where the development process and reviewing process are supported by separate Git and mailing lists, there is no explicit link between the reviews of a patch and its corresponding commit. Here, we analyze whether, despite the three challenges of the introduction, the three linking techniques are still able to link a commit to at least one of its patch versions. If so, the techniques could be used to retrieve details and comments about the design rationale of a commit from corresponding emails.

Approach: We applied the three techniques to find the correct (email, commit) pairs. We then calculate precision, and **relative** recall of a representative sample of 384 (email, commit) pairs, one such sample per technique (all percentages need to be interpreted as having a 95% confidence level ± 5% confidence interval). Finally we also calculate the overlapping results between the three techniques. The results of precision, relative recall and F-score are listed in Table 5.2.

Findings: **The checksum-based technique has the highest precision in linking commits to emails.** The checksum-based technique detects 216,033 (email, commit) pairs, while the plus-minus-line-based technique detects 531,381 and the clone-detection-based technique (CCFinderX) detects 168,408 (after filtering with an RSA threshold of 0.4). After manually validating 384 random samples, we obtained a precision of 97.92% for the checksum-based technique (376/384), 85.16% for the plus-minus-line-based technique (327/384) and 81.77% for the clone-detection-based technique (314/384).

The checksum-based technique has the highest precision since it is the most strict, which means that it is very hard to accidentally find two almost identical changes (chunks) to the same file. The heuristic to match a commit to an email patch as soon as they share one

Table 5.2 Evaluation of three techniques of linking (email, commit) pair.

technique	precision	relative recall	F-score
Checksum	97.92%	47.68%	0.64
Plus-minus-line	85.16%	85.69%	0.85
Clone-detection	81.77%	37.40%	0.51

common chunk seems to work well. On the other hand, the clone detection approach makes the most errors since it is the most liberal of the three.

The plus-minus-line-based technique has the highest relative recall. Out of the 992 true positive (email, commit) pairs identified across the three techniques (union of unique true positives after manual analysis), 473 are detected by the checksum-based technique, 850 by the plus-minus-line-based technique and 371 by the clone-detection-based technique. The relative recall for each technique is 47.68%, 85.69% and 37.40% respectively. Hence, relative recall-wise the plus-minus-based technique is almost twice as good as the other two techniques.

We also found that between 16.98% and 19.23% of the techniques' true positives mapped a commit to the last available version of a patch, while more than 80% mapped the commit to earlier versions (which are more different and hence harder to match). The low performance of the checksum-based technique is due to activities like cherry-picking or modifications below the chunk-level. Since checksums are calculated per chunk (i.e., per changed file), a patch that is modified before being committed just to fix a typo or some other non-whitespace related reason will yield a different checksum and hence not be linked correctly. This suggests that the checksum-based technique likely needs to be refined below the chunk level.

The plus-minus-line-based technique has the highest F-score in linking (email, commit) pair. Since the highest precision and relative recall go to different techniques, to verify which technique is the best one overall we compute the F-score for each technique, which is a combination of precision and recall [158]. As expected based on the recall results, the plus-minus-line-based technique has the highest F-score (0.85), and hence works best for linking commits to emails. It seems a good compromise between strictness and granularity. Being less strict means that it can avoid false negatives while its fine granularity still does not concede too many false positives.

The plus-minus-line-based technique can link the majority of (email, commit) pairs. Figure 5.7 shows the overlap of results of the three techniques. Only 3.83% and 7.16% of the pairs are identified uniquely by the checksum-based technique and the clone-detection-based technique respectively, whereas the plus-minus-line-based technique is the most effective technique for linking commits to email patches.

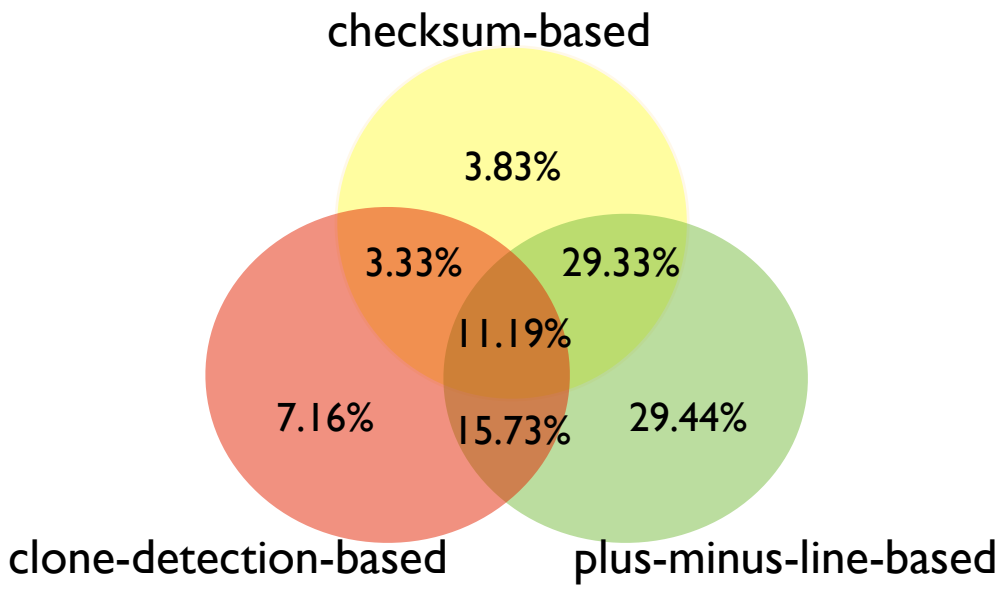


Figure 5.7 Overlap of (email, commit) pairs detected by the three techniques.

For linking emails to commits, the plus-minus-line-based technique has the highest relative recall and F-score, and it covers most of the correct (email, commit) pairs.

RQ2: Can emails containing different patch versions be linked accurately to each other?

Motivation: In the context of low-tech reviewing environments, the reviewing process consists of replying to emails containing a submitted patch, sent to a developer mailing list. However, sometimes the contributor may start a new thread to submit a new version of a patch, causing multiple threads to exist that talk about the same patch (we call this set of threads a super-thread), yet do not contain an explicit link to each other and to the final commit. Hence, people may regard one thread of a super-thread as the sole reviewing thread of a certain patch and ignore other related reviewing threads, which will cause underestimation of the actual reviewing process and miss important design decisions.

Approach: In order to recover the reviewing super-threads, we apply the three linking techniques to search the emails containing similar patches across different threads. We then compute the precision and relative recall of the results of each technique, as well as the overlap between techniques (just like for RQ1). Contrary to email-to-commit linking, we can also provide real recall based on threads containing annotations in their subject. The results are listed in Table 5.3. For CCFinderX we obtained a different RSA threshold (0.6) than for RQ1.

Findings: **The checksum-based technique has the highest precision.** The checksum-

based technique detects 3,020,582 (email, email) pairs, while the plus-minus-line-based technique detects 4,565,857 and CCFinderX detects 6,124,303, respectively. We again randomly select 384 samples from the results of each technique. After manually validating the samples, we obtained a precision of 86.98% (334/384) for the checksum-based technique, 51.82% (199/384) for the plus-minus-line-based technique, and 70.83% (272/384) for the clone-detection-based technique.

The checksum-based technique again has the highest precision, for similar reasons as for RQ1. However, the clone detection technique and (especially) the plus-minus-line-based techniques dropped in precision. The explosion in number of patches available for linking makes the plus-minus-line-based technique select incorrect patches.

The plus-minus-line-based technique has the highest relative and real recall. Out of union of 796 true positive (email, email) pairs, 424 are detected by the checksum-based technique, 547 by the plus-minus-line-based technique and 148 by the clone-detection-based technique. According to our definition, the relative recall of the checksum-based technique is 53.27%, of the plus-minus-line-based technique 68.72% and of the clone-detection-based technique 18.59%.

For email to email linking, we also have an actual ground truth based on a convention in Linux kernel reviews where the subjects of patch versions include tags like “V2” or “v3”. Starting from the 463,697 subjects of the first email of each thread, we used regular expressions to remove version tags like “V2” or “v3”, then counted how many subjects become identical (only difference was the tag). Only 1,912 subjects (i.e., 0.41% of the threads) could be linked this way. Although only few experienced contributors follow this convention, we randomly selected 384 samples with confidence level of 95% and confidence interval of 5% as ground truth, then checked how many of them are detected by each technique.

We found that the recall of the checksum-based technique is 76.83%, of the plus-minus-line-based technique 80.47% and the recall of the clone-detection-based technique is 23.44%. Although the values are slightly higher than the relative recall, it follows the same trend. As in RQ1, the plus-minus-line-based technique still has the highest (**relative**) recall, but with a lower performance than for (email, commit) pairs. The checksum-based technique does better than for email to commit pairs, while the clone-detection-based technique loses half of its relative recall compared to email to commit pairs. Note that, similar to RQ1, most of the true positive results were relatively hard to match: less than 8.72% of the matched pairs of each technique consisted of patch versions sent in the same thread, all others were between patch versions in different threads!

The checksum-based technique has the highest F-score. Since the highest precision

and relative recall again belong to different techniques, we compute the F-score for each technique, similar to RQ1. The highest F-score goes to the checksum-based technique.

The checksum-based and the plus-minus-line-based techniques are complementary. Using the ground truth of 796 (email, email) pairs for relative recall, we can see (Figure 5.8) that 21.08% of them are detected only by the checksum-based technique, while 39.37% could only be found by the plus-minus-line-based technique compared to 4.99% by the clone-detection-based technique. The combination of the checksum-based and the plus-minus-line-based techniques together take up more than 95% of the ground truth.

To better understand the performance of each technique, we randomly picked 10 pairs from the results of each technique that are only detected by itself to analyze their characteristics: The patches only linked by the checksum-based technique, typically range from 15 LOC to 3,334 LOC, with an average size of 427.6 lines, compared to 3 LOC to 541 LOC (99.5 LOC on average) for the plus-minus-line-based technique and 45 LOC to 7,844 LOC (1,196.7 LOC on average) for the clone-detection technique. This difference stems from the fact that the checksum-based technique matches full chunks instead of fine-grained lines or groups of tokens. Furthermore, the plus-minus-based technique only matches pairs of patches who have a majority of lines in common, which makes it hard for patches with uncomparable size or large patches to obtain a match. Finally, the very large size of clone-detection-matches does not just come from the minimum size threshold used by clone detection tools, but rather from the many false positives amongst small clones. Generic patch lines consisting just of an assignment or a for-loop will be matched to many emails incorrectly, while large sets of lines or whole files containing common lines have a much higher chance of being uniquely distinguishable and hence map only to the right patch.

For linking emails to emails, the checksum-based technique overall performs the best, whereas it requires the plus-minus-line-based technique to obtain a higher recall.

RQ3: What are the characteristics of the reviewing history in a low-tech reviewing environment?

Motivation: Now that we have successfully evaluated algorithms to link commits to their earlier email patches, we can now (for the first time) study low-tech reviewing environments from a quantitative point of view. Similar to studies on modern reviewing environments [40][48][159], one can now study the following questions: How long do email-based reviews take? How many reviewers and reviews are there? Since this paper is the first to trace back from a commit to the earliest emails containing patch versions, we also try to understand why some patch versions are spread across multiple threads. Is this a bad situation for a

Table 5.3 Evaluation of the three techniques for linking (email, email) pairs.

technique	precision	rel. recall	F-score	real recall
Checksum	86.98%	53.27%	0.66	76.83%
Plus-minus-line	51.82%	68.72%	0.59	80.47%
Clone-detection	70.83%	18.59%	0.30	23.44%

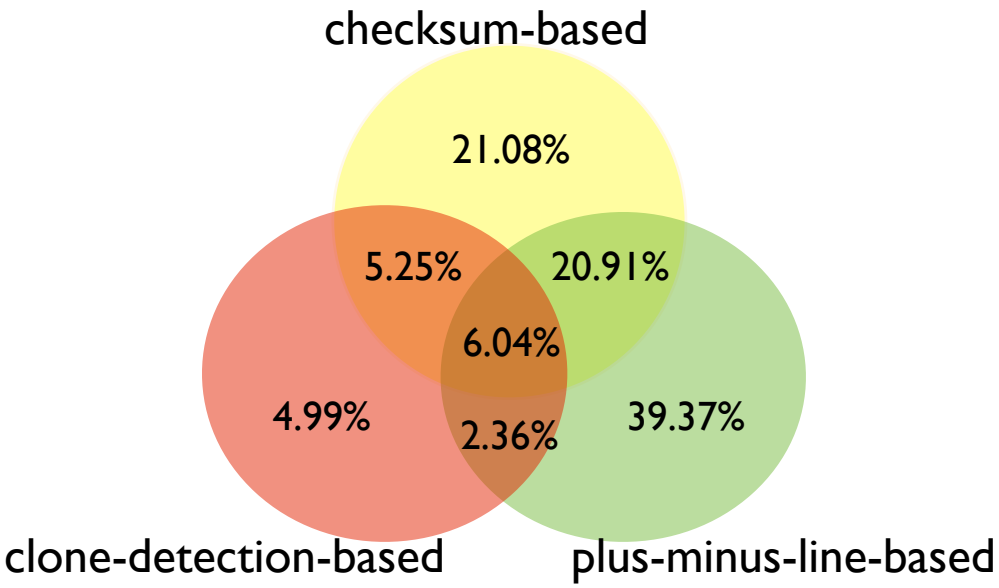


Figure 5.8 Overlap of (email, email) pairs detected by the three techniques.

submitter (lower acceptance rate?) and reviewers (waste of time to review later versions?), or rather a recipe for success (higher acceptance rate?)? What kind of patches (bug fixes, large vs. small, etc.) typically have multiple versions?

Approach: To answer the above questions, we combine the best and most complementary techniques for linking emails to emails (the checksum-based and the plus-minus-line-based) to obtain a linking technique capable of obtaining a high precision and recall for recovering the super-threads of commits on the full data set, not just on a sample of 384 pairs.

Manual analysis of the resulting super-threads showed several infeasible matches with emails more than ten years apart. To filter out such matching results, we only keep super-threads with a time duration below the upper tail (ut) of the data set according to the following formula [159]:

$$ut = (uq - lq) * 1.5 + uq$$

Among them, uq indicates the 75% quartile, while lq indicates the 25% quartile of the data as shown in Figure 5.9. The data with value higher than the upper tail uq are considered to

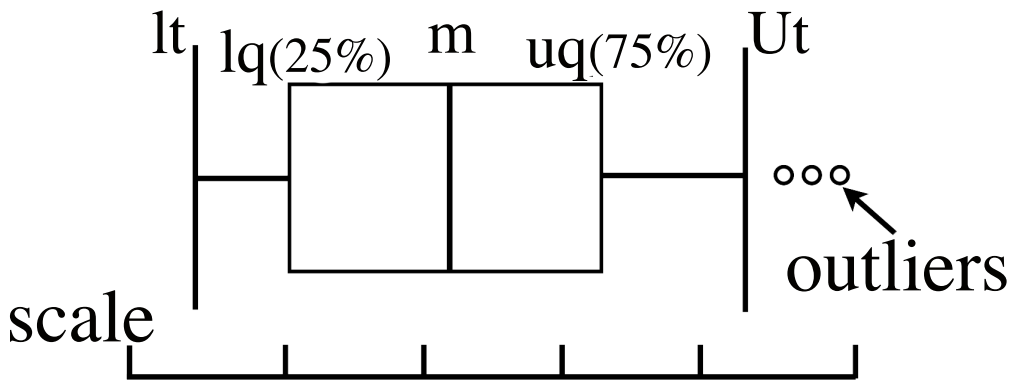


Figure 5.9 Boxplot indicating how to remove the outliers in our data.

be outliers and we remove them. Note that (contrary to RQ1 and RQ2) this RQ considers both patches that are accepted (linked to a commit) as well as those that are rejected (no corresponding commit).

Findings:

Qa) *How far back does the reviewing history of a patch version go?*

Around 25% of the MM patches has a previously "hidden" reviewing history of more than four weeks. We analyzed the time duration of the MM super-threads, i.e., from the very first design discussion until the last discussion of a patch (possibly across multiple threads in case of MM). Table 5.5 shows that most of the super-threads consist of only few patch versions (Mean. value is 3.172), but can last a long time (Mean. value is 21.341 days). More than 25% of the patches have a reviewing time that is 4.5 weeks longer than considered by researchers thus far [100].

Qb) *What kind of patches undergo multiple patch versions?*

Patches evolving across multiple versions are larger and affect more files than those with a single version. The patches from threads of type MM (especially) and MS have higher values for "size", "spread" and "spread_subsys", as shown in Table 5.4. This indicates that the patches undergoing multiple versions tend to be larger and more complex, and hence need more attention before being integrated. Surprisingly, such invasive patches seem to feature especially in single threads, rather than multiple ones. As we will see below, this means that they are still integrated relatively quickly, whereas the patches that need more versions tend to be slightly less invasive. A Kruskal-Wallis test with post-hoc tests verified the significant difference.

Table 5.4 Average value of characteristics in different types of threads (including rejected patches).

	metric name	MM	MS	SS
Review	thr_volume	3.838	6.051	3.533
	nr_reviews	1.046	1.936	1.165
	review_time (day)	1.932	3.022	2.271
	response_time (day)	0.871	1.131	1.030
	first_response_time (day)	0.801	1.215	1.010
Patch	size	81.660	146.100	25.430
	spread	2.398	3.811	1.016
	spread_subsys	1.387	1.750	1.003
Other	acceptance	46.05%	43.97%	23.26%
	bug-fix	49.94%	36.72%	31.10%

Table 5.5 Time duration (#days) of the super-threads of type MM.

	time duration	# of patch versions
Min.	0	2.000
1st Qu.	2.687	2.000
Median	10.061	2.000
Mean	21.341	3.172
3rd Qu.	32.923	3.000
Max.	107.524	108.000

Qc) *What kind of patches undergo multiple threads?*

Kernel developers use multiple threads if too much time has passed since the previous patch version. We compared the time distribution of the interval between two successive threads to that of two successive patch versions within one thread. The result is shown in the boxplots of Figure 5.10. We can see that the time interval of threads is much longer than that of patch versions. This seems to confirm the intuition that people typically start a new thread when too much time has passed since the last review or version of a patch, whereas they would continue the same thread otherwise. A Mann-Whitney test with as null hypothesis "no difference between the average of both time distributions" obtained a p-value <2.2e-16, which confirms that the differences are statistically significant.

Threads of type MM especially consist of bug-fixes. Out of all MM threads, 49.94% are bug-fixes, compared to 36.7% for MS and 31.10% for SS threads. The pair-wised Mann-Whitney test show that MM has no significant difference with MS, but that MM and MS are significantly different from SS.

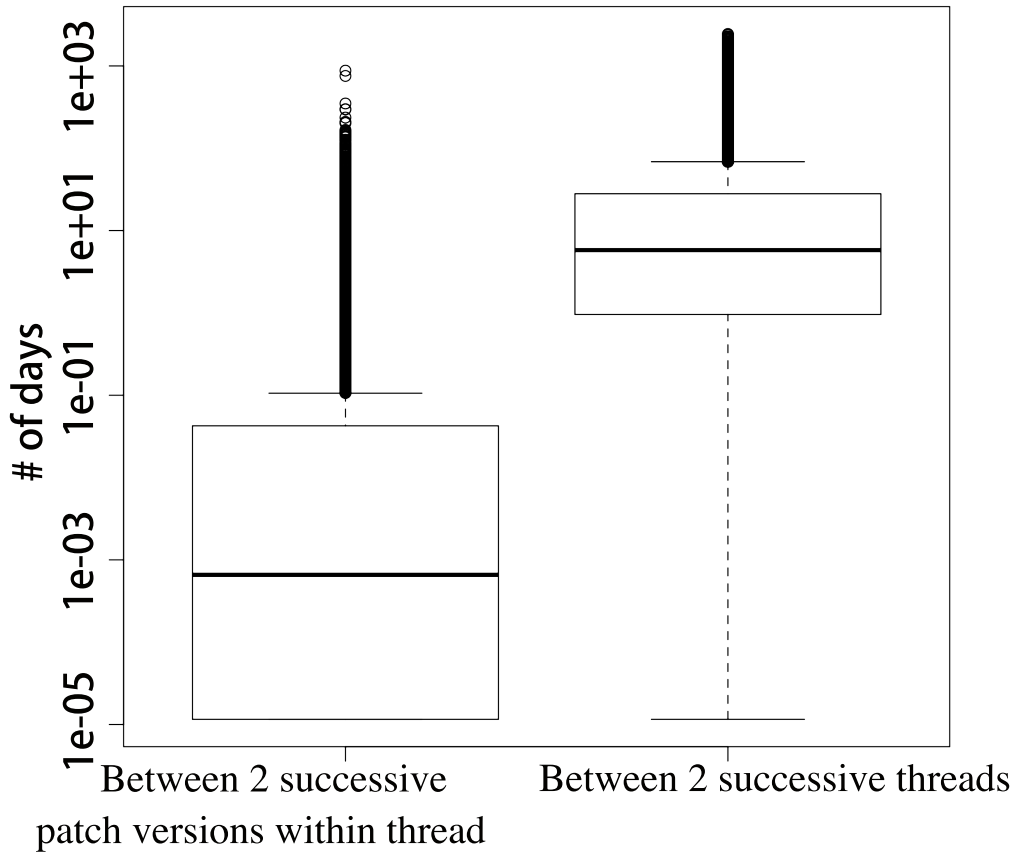


Figure 5.10 Boxplot of average time interval (#days) between two successive patch versions/threads of super-threads.Given the log-scale, a value of 1e-05 in fact denotes zero.

On the one hand, this finding seems surprising, since one would expect bug-fixes to be smaller and hence require less discussion. On the other hand, bugs might be risky to fix, and hence require care and thorough reviews.

Qd) *Do reviewers lose interest in multi-version patches?*

Patches of MM and MS threads involve more discussion than SS threads. We compared the number of reviews discussing a patch for the three different types of threads. The value of "thr_volume" of MM and MS patches is higher than for the SS, which means that if a patch needs to undergo one or more additional versions reviewers seem to discuss more about it and provide more constructive comments to help improve it to be accepted. The amount of reviewing hence does not suffer from having multiple versions of a patch. A Kruskal-Wallis test with post-hoc tests showed that the three groups are different from each other.

Patches of type SS and MS have fewer number of reviews. SS and MS patches

receive the most receives (nr_reviews), and hence take more review_time as well. Hence, it seems like patches evolving across multiple versions attract fewer reviews. A Kruskal-Wallis test with post-hoc tests showed that the difference is significant. One possible explanation could be that MM patches received the majority of their reviews early on, with later patch versions receiving more focused reviewing.

Reviewers are more eager to review MM threads. Although we did not find a statistically significant difference between the distributions of metrics "response_time" and "first_response_time" for SS and MS, we found that MM threads significantly take less time before the first review. Reviewers seem to consider such patches as having a higher priority than other types of superthreads.

Qe) *Do multi-version patches have a lower chance of acceptance?*

No, threads of type MM in fact have a higher acceptance rate. In Table 5.4, we have marked a super-thread as accepted if at least one of its patch versions could be linked to a commit. We found that the threads of type MM have the highest acceptance rate (46.05%), followed by MS patches with 43.97% and SS patches with 23.26%. A Kruskal-Wallis test for comparison of the characteristics among the three groups shows for all 3 patch-related metrics a p-value <2.2e-10, meaning that at least one is significantly different from the others. Subsequent post-hoc tests show that all three are significantly different from each other. In other words, being asked by reviewers to write a second or later patch version is not a guarantee for success, but at least a good sign: reviewers see potential and really want to work on the patch.

5.5 Threats To Validity

Threats to external validity concern the fact that we have only studied one large open source system. The Linux kernel is a prime example of a large open source project that uses its mailing list as reviewing environment. We cannot automatically generalize our findings to other projects, potentially supported by different reviewing mechanisms. Furthermore, since Git is a flexible version control system that can be used in various setups, we also need to take care of extrapolating our findings to other projects using Git or other version control systems. Hence, more case studies on other projects are needed.

Threats to internal validity concern the relationship between treatment and outcome. Since we did not study causal relationships, but quantitatively analyzed the characteristics of the reviewing history, we do not consider threats to internal validity.

Threats to construct validity concern the relation beween theory and observation. We link the different threads belonging to the same super-thread together. However, the three techniques that we applied are not perfect (precision and relative recall below 100%). Although we use the complementary results of two techniques to improve the precision and recall, and remove the outliers in the final results, the super-threads may not be the real super-threads, due to false positive noise. However, given the amount of email and commit data that we consider, we believe that the impact of noise is tolerable and that the resulting trends are valid.

5.6 Related Work

Our work is mainly related to previous studies on the code review process of on open source projects, traceability, mining unstructured log data and clone genealogies.

Rigby et al. [136] analyzed the code review process in the Apache open source system. They found that small, independent, complete patches are faster to be accepted. In our paper, we found that the patch versions in a super-thread are examples of patches that take longer to be accepted. Since they seem to be larger and more invasive, this confirms Rigby et al.'s findings

Bettenburg et al. [51] found that accepted contributions are on average 3 times larger than rejected contributions, however, we did not study this phenomenon here.

Rigby et al. [134] studied the peer review process of a traditional inspection of a Lucent project and six open source projects. They found that smaller changes take shorter review time and the time from the start of the review to the end of the review was on the order of weeks. Both of the findings are verified in our paper.

Mcintosh et al. [115] studied the impact of modern reviewing environments (Gerrit [95]) on software quality. The impact is assumed to be due to a large amount of freedom that reviewers have when reviewing at their moment of choice via a website. It is not clear if low-tech reviewing environments based on emails have the same issues.

Jiang et al [100] studied the integration process of the Linux kernel project. However, they only considered the reviewing history inside the current thread and did not consider the possibly "hidden" super-thread reviewing history. To our knowledge, the current paper is the first to study the whole patch reviewing process across multiple threads in a low-tech reviewing process, we also analyze the impact of different thread types on the acceptance probability and the characteristics of different thread types.

There has been a lot of research on traceability between source code and software-related

documents [38][81][82]. However, our work is the first to trace the commit back to its origin to figure out its rationale. Bettenburg et al. [55] proposed an approach to link email discussion text to source code fragments based on clone detection technique. This is one of the three techniques in our work. Bettenburg et al. [52] developed a tool for mining the structural information from natural text bug reports. Bacchelli et al. [41] proposed an approach to classify email content into different source code artifacts at the line level. Since Git patches have a very rigid format, we did not need to use there techniques to identify the Git patches in emails.

Our work is also related to the research on the vast body of code clone genealogies during the software evolution process [53][72][104][110], since those also track code (changes) across time. Our work combines clone genealogies with the reviewing process in a low-tech environment to help developers and reviewers understand the design rationale of patches.

5.7 Conclusion

Tracing the reviewing process of a patch to figure out its rationale plays an important role in software maintenance. However, a low-tech reviewing environment is still commonly used in many open source systems. In our study of the Linux kernel project, the reviewing and development process are separated and furthermore, a patch may vary across multiple versions and threads. In this paper, we propose an approach to recover the full lifecycle from commits to emails and analyze the characteristics of different types of reviewing threads. We found that the plus-minus-line-based technique is the best to link (email, commit) pairs, while the combination of the plus-minus-line-based technique and the checksum-based technique works best for linking (email, email) pairs.

As a first application of the recovered email to email links, we quantitatively analyzed the reviewing process in a low-tech environment and found that around 25% of the patches have a reviewing history of more than four weeks. Patches with multiple versions seem to be larger and impact more files, while they are spread across multiple threads if they contain bug fixes and long time has passed since the previous patch. Having to submit additional patch versions is not a disaster, since relatively more such patches are accepted than for patches with just one version, and reviewers keep on being interested in subsequent versions.

Our work opens up the possibility to compare low-tech reviewing environments head-to-head with modern reviewing systems. We consider this to be our future work.

CHAPTER 6 ARTICLE 3: EMPIRICAL CASE STUDY OF HYBRID INTEGRATION APPROACH

Yujuan Jiang, Josh Chiang, Budhai Roy, Bram Adams

Abstract

The traditional "Branching" integration approach forces a major problem: "Integration Conflicts". Such conflicts are caused by long term isolation, and take developers effort and time to fix. A new integration technique, "Toggling", came up as a solution by hiding features behind "if" conditions to solve this problem, which allows teams work on different features on only one branch. However, toggling causes controversial arguments about its tendency toward technical debts as well as code complexity. So far, as many IT companies like Facebook and Google have adopted toggling in their systems, many companies are trying to migrate to this new technique. However, changing to this new technique from scratch might be risky because of lack of experience and guidelines. We study a project that adopted a compromise combining branching and toggling together (hybrid). We look into the difference between the current hybrid structure and the original pure branching structure from two perspectives (Integration Effort and Productivity), aiming to understand if this hybrid structure works better than branching. We found that the hybrid approach can help reduce integration effort and improve productivity compared to the branching. Hence, this compromise might be a good choice for companies that want to adopt toggling in their development process.[1]

6.1 Introduction and Motivation

Branching is an integration technique of that allows developers to work on a copy ("branch") of a repository ("trunk"). Developers can integrate (merge) any change task to trunk when ready. Hence, Branching enables different teams to work on different features in parallel isolatedly. This isolation helps prevent disturbing each other, yet also causes one of the major issues of branching, which are so-called "Integration Conflicts".

This is because after working separately for a long time, the original context where developers started might have been changed by other developers. Then, developer need to spend time

[1]Paper submission pending on agreement by company.

and effort to fix these conflicts in order to merge new code changes. For example, developer A uses a public variable x in his new function, while developer B changes the permission of variable x to private. After developer A integrates his change into trunk, it will fail to compile the new version because the variable x is no longer accessible. Now developer A needs to discuss with B about how to fix this problem, which could take a long time and require coordination between different teams.

Toggling (also called "switches", "flags" or "flippers") emerged in order to overcome the "Integration Conflict" problem typically associated with branching. Different teams will work on only one branch, using individual conditional statements ("toggle") to isolate different tasks ("feature"). For example, if a new feature is under development, the whole code block is toggled by an if statement "if(featureIsReady)". If we set the toggle "featureIsReady" to be true, the whole feature code block will be visible and can be tested, otherwise if the toggle is false, this code block will have no impact on the whole project, no matter if it is ready or not. Once this feature is stable in a couple of releases, the conditional code will be removed and the feature will become permanent.

Theoretically, toggling should help overcome the isolation problem of branching, yet at the same time it has been criticized for its own problems such as introducing technical debt and increasing complexity. Currently, many IT companies such as Facebook, Google and Flickr have adopted toggling in their development to replace older branching structure, in order to deliver new features sooner to their end users. However, so far there has not been any quantitative evidence yet about whether adopting toggling really improves the integration efficiency. Furthermore, as this is a rather new phenomenon, for those companies trying this new technique there is a lack of good practices and prior models to follow. As such, it might be too risky for a commercial company to completely replace traditional branching.

Given this challenging practice, a compromise might be recommended at the beginning. Technically, one can combine branching and toggling together in a hybrid structure, where toggles are used in a branching structure. Eventually, if toggles really work better, branches can be removed gradually and the whole project can migrate to full toggling. If this hybrid structure also improves integration efficiency and helps avoid integration conflict, it is a good start for companies that want to try this new integration technique.

In this paper, we empirically study a project that adopts the hybrid structure (combination of branching and toggling), aiming to understand if this compromised solution really works. We will compare the two integration approaches (hybrid and pure branching) from the following two perspectives:

RQ1) *Does hybrid structure take less integration effort than branching?*

It turns out that the hybrid structure helps reduce more integration effort than branching, and takes less time to finish a complete feature.

RQ2) *Does hybrid structure improve productivity?*

It turns out that hybrid structure helps accept more production code and abandons less. The hybrid structure also helps reduce the number of revisions a patch needs to go through before being accepted. Additionally, the hybrid approach integrates more but smaller commits, while the whole contribution is larger than for a branching approach.

6.2 Related Work

Continuous Integration. Continuous Integration (CI) [62] [142] is a practice of release engineering that encourages developers to integrate any new change they make into the central code base as soon as possible. Along with this concept, a lot of practices have emerged, such as automated unit tests [29] [161], automated build server (e.g., Jenkins) and DevOps [46], in order to improve the integration efficiency. There are more and more researchers and workshops that focus on how to produce and deliver good product faster and better [21] [32]. However, as far as we know, our work is the first quantitative comparison of toggling & branching integration techniques.

Branching & Toggling. Branching and toggling are two integration techniques. Branching avoids developers from disturbing each other, but does not deal well with "Integration conflicts", while toggling allows team to work on the same branch, but increases code complexity and technical debt [7] [8]. Our work is the first to compare the two integration technique in practice from multiple dimensions to figure out if there is a compromise between these two techniques.

Key Performance Indicator (KPI).

A Key Performance Indicator (KPI) is a management tool that is used to measure the performance of a developer or the effect of development activities. KPIs are not only used for business and marketing, in IT industry field they are also widely used to supervise the health of a community or a system [127]. Fotrousi et al. [78] conducted a literature review about 34 relevant studies about KPI. They gave recommendations to practitioners about how to use KPIs in the management of an ecosystem and showed researchers the gap between

theory and practical use of KPIs. Our work is the first to use KPIs to quantitatively analyze the impact on development performance of different integration techniques.

Rotella et al. [140] at Cisco studied how quality metrics work and how to implement applicable metrics to meet quality requirement. Our work is the first one to use quantitative metrics to compare two different integration techniques and figure out which one works better in practice.

6.3 Approach

6.3.1 Selecting Case Study

Project P of company S (anonymous names) was launched in May 2014. Project P is a cloud-based data analytics software, containing more than eight million lines of code contributed by more than six hundred developers. This project started with a branch-based integration approach in their development process, and as of May 2015 started adopting "Feature Toggling" to improve the integration and testing efficiency. As removing all branches would be a large step to take, they do not eliminate all branches but adopted a hybrid structure (combination of branching and toggling), where there still exists a trunk branch with toggles being used to determine whether release a new feature to next version.

Figure 6.1 shows the structure of this project of all code repositories before and after adopting toggling in development process. The central white node is the trunk, which houses the whole product's code base. The red nodes are those sub-branches that contribute to the central branch.

6.3.2 Feature Contribution

In order to check if a hybrid structure works better than a pure branching integration approach, first we introduce the concept of "Feature Contribution". As shown in Figure 6.2, when a new feature is started being developed, developers need to fork a feature branch from trunk (sync-up). Once this feature is completed, developers need to integrate all changes on this feature branch back to trunk (merge). Before the final merge, there could be additional sync-ups from trunk to feature branch, in order to keep the branch synchronized with trunk. A "Feature Contribution" consists of all commits from the first sync-up until final merge back to trunk. It corresponds to a contribution cycle of submitting a complete feature.

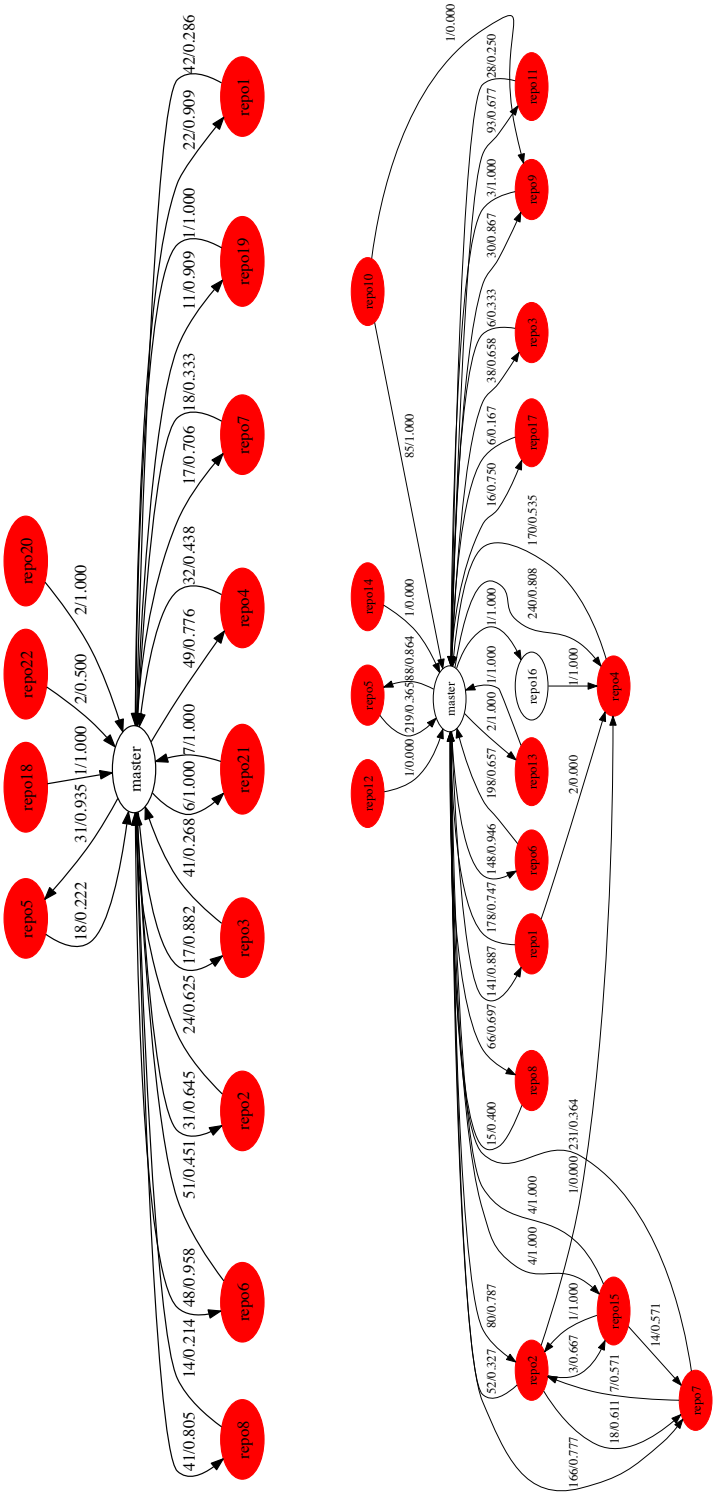


Figure 6.1 Structure of before (above) /after (below) toggling.

6.3.3 Data Collection

We collected the data of this project from May 2014 until September 2015, mainly from three sources, including the build information from the CI server Jenkins (e.g., the number of releases and the release date), review information from the Gerrit reviewing system (e.g., the number of patch versions a commit went through, and whether a commit is accepted or abandoned at the end) and development information from Git (e.g., the number and date of commits).

6.3.4 Key Performance Indicators

In order to compare the two techniques, we used a group of KPIs (Key Performance Indicator) that are used to measure performance and production activities in this company, after discussing with engineers of this company. Finally, we nailed down two main KPIs covering different dimensions (shown in Table 6.1). These KPIs are the most concerned ones from the industry perspective.

Integration Effort indicates how difficult it is to integrate a commit into the trunk. Practically, when a branch is merged in another branch seamlessly, the branch's merge commit should be empty. On the contrary, if a commit does not fit in perfectly, and causes integration conflicts, developers need to manually fix the conflicts in the merge commit. The more adaptation needs to be made, the larger this merge/sync-up commit will be. We define the integration effort as sum of lines of code of all sync-up and final merge commits during a "Feature Contribution" cycle (6.2):

$$Integration\,Effort = \sum(ESyn) + EFm \tag{6.1}$$

Table 6.1 KPIs and corresponding decsription.

Dimension	KPI	description
Integration Effort	Integration Effort	Sum of LOC of sync-ups and merge commits of one feature contribution.
	Normalized Effort	Integration Effort per Contribution LOC.
	Integration Time	# Days a feature contribution cycle lasts.
Productivity	Volume	# abandoned/accepted patches per calendar week (across all branches vs. only master branch).
	Contribution Size	Size of individual commits & complete contribution.
	Revision Number	# Revisions one single commit goes through before being accepted/rejected.

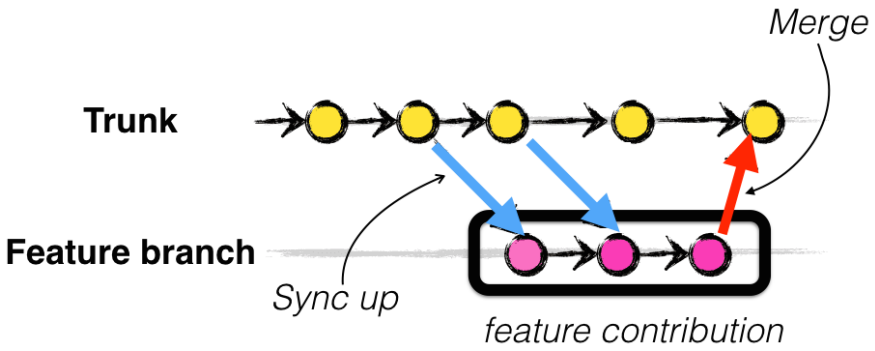


Figure 6.2 Process of contributing a feature (Blue arrow indicates sync-up and red indicates merge).

Effort is the total integration effort of contributing a feature, while ESyn means the effort of sync-ups and EFm means the effort of final merge.

Furthermore, we also normalize the integration effort by the size of the whole contribution, to reduce the impact of large features.

$$Normalized\,Effort = \frac{Integration\,Effort}{Size\,of\,Complete\,Contribution} \tag{6.2}$$

Besides code, we also consider the time length of a feature contribution cycle ("Integration Time"). This is defined as the number of days from the first sync-up until final merge back to trunk.

$$Integration\,Time = \#\,of\,Days\,of\,a\,Feature\,Contribution \tag{6.3}$$

Productivity: In the development process of project P, a commit needs to go through peer review before being merged into trunk branch. Patches are submitted and reviewed, once integrated they become a commit. If this patch passes the automatic testing/building system (i.e., Jenkins), it will be integrated into trunk as a commit. Otherwise, the developer of this patch needs to still work on this patch according to feedback from reviewers and later submit another version of this patch. A commit may experience multiple patch versions before being accepted.

First, we use the number of accepted and abandoned commits per calendar week to measure productivity. More commits being accepted, or on the other hand, less commits being abandoned indicates a high productivity.

$$Productivity = Accepted \,|\, Abandoned\, Commits\, per\, Calendar\, Week \tag{6.4}$$

We also use the size of single commits as well as the complete feature contribution (the real production code integrated in a feature contribution cycle) to measure the productivity of the two approaches.

$$Contribution\, Size = \sum LOC\, of\, Single\, Commit\, or\, Complete\, Feature\, Contribution \tag{6.5}$$

Additionally, we use the number of patch versions to indicate the efficiency of acceptance or abandonment.

$$Revision\, Number = \#Patch\, Versions\, per\, Commit \tag{6.6}$$

6.4 Case Study Results

6.4.1 Integration Effort

Motivation: In the ideal situation, a commit should be integrated into trunk without any additional effort. Otherwise, it would take developers effort and time to fix these inconsistencies. Thus, a good integration approach should help developers avoid the potential of integration conflict and reduce the effort needed to fix a conflict, as well as integrate or give up a feature contribution as soon as possible.

Approach: We compare the integration effort and normalized effort for both branching and hybrid structure, as well as the number of days a feature contribution cycle lasts.

We visualize the distribution with beanplots, which is a visualization tool based on boxplot but with extra density information.

Findings: A hybrid structure takes less effort to contribute a new feature than branching. Figure 6.3a shows the total integration effort of branching and hybrid structure respectively. We can see that the raw integration effort of branching (with a median value of 16 lines of code) is higher than for hybrid structure (median of 13). However, the t-test did not show significant difference. After normalizing this effort by the size of the contribution size (Figure 6.3b), we observe that branching still takes more relative integration effort (with a median value of 0.027 effort per line of churn) than the hybrid structure (median of 0.007). This time, the t-test shows a significant difference between both distributions. A

further effect size test shows that this difference is a medium one, which indicates that the integration effort per code line would be reduced by toggling.

This seems to confirm the intuition that toggling enables developers to submit new changes sooner such that the integration conflicts are less common or easier to fix because of the shorter isolation time.

Toggling takes the same time to merge a commit as branching. Figure 6.4 shows the time it takes branching/hybrid structure to merge or give up a completed feature contribution. The black bean represents the distribution of acceptance while the grey bean represents the distribution of abandonment. We can see that hybrid structure takes longer to merge/abandon a contribution than branching. It takes toggling 1.07 days to abandon and 1.12 days to merge a contribution (in terms of median value), while branching takes 0.81 days to abandon and 0.79 days to merge a complete contribution (median). However, a statistical test did not show a significant difference for the time distribution, so basically the time of an integration does not differ too much between these two integration approaches.

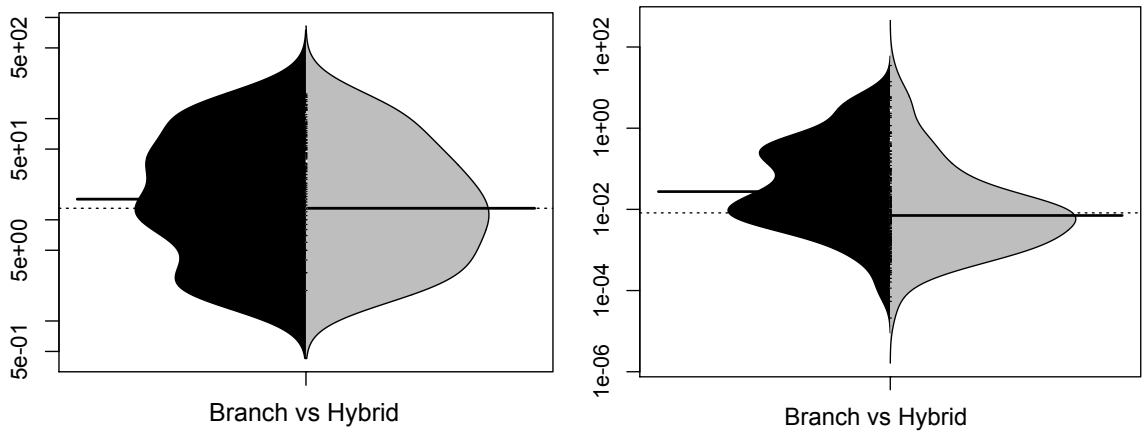
This could be because (1) with toggling, the average contribution size increases so the time it takes to complete a feature contribution also increases (we will indeed see such an effect later on); (2) company S has been using this hybrid approach only for a short time, it could still take developers time to get used to the new structure and technique. A follow-up study should be conducted to verify whether the integration time is reduced after the new structure is more steady.

Hybrid-toggling takes less effort to integrate a new contribution.

6.4.2 Productivity

Motivation: Productivity is a very important indicator for an industrial company, since it hooks up directly with the market. The higher the productivity is, the faster the new features and bug-fixes are sent to end users. Boiling down to low-level details, high productivity means fewer reviews/revisions a commit goes through to get integrated, less waste (abandonment) of production code, less time to review a commit and larger contribution size. A good integration technique should help achieve this high productivity in practice.

Approach: We count the number of accepted/abandoned commits per calendar week for both techniques. In order to control the numbers we also normalized by the total number of commits. Additionally, we applied the same process on trunk only to see if the same trend still holds.



(a) Total effort (b) Total effort normalized by contribution size

Figure 6.3 Integration effort (original and normalized by contribution effort) (both log-scaled).

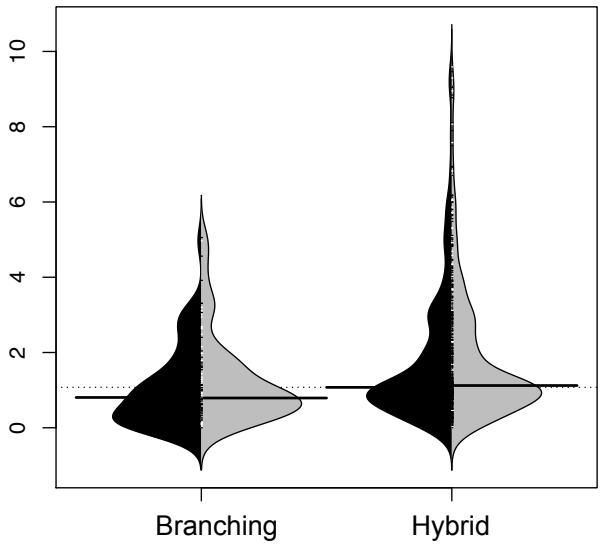


Figure 6.4 # days to accept/reject a contribution (black: abandoned, grey: accepted).

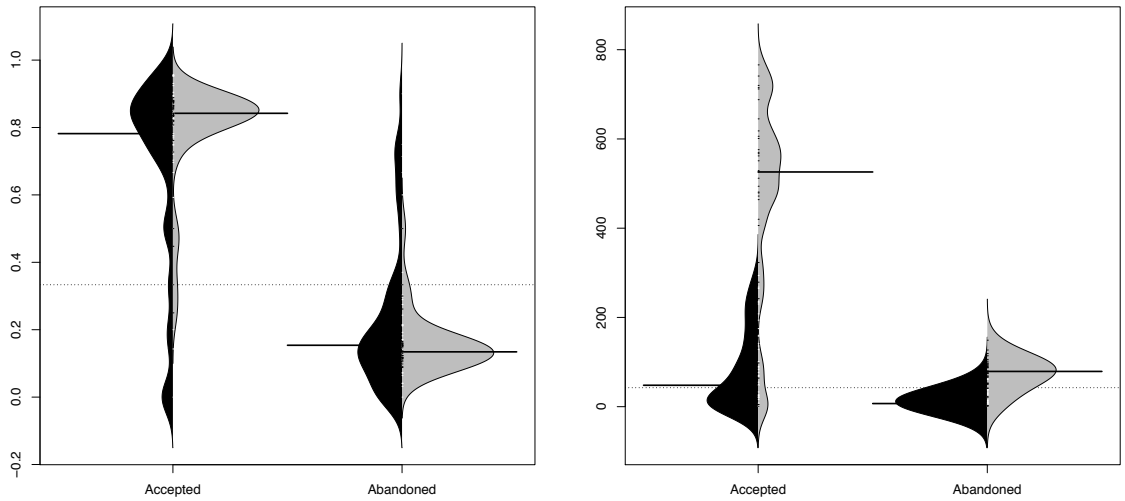
Furthermore, in order to understand the deep reason of the difference, we also looked into the size of accepted/abandoned commits, and the number of patch revisions an accepted commit went through before being abandoned/merged.

Findings: A hybrid structure accepts more commits and abandons less commits than branching. Figure 6.5b indicates that the number of accepted/abandoned commits per calendar week are both higher for toggling than for branching. After being normalized by the total number of commits, we found that hybrid structure accepts a higher percentage of commits and abandons less than branching (Figure 6.5a). This also holds when considering only trunk (Figure 6.6). This indicates that toggling can increase the velocity of commit integration, because toggling enables developers to integrate their code whenever they want. This makes the integration process and the feedback goes faster, such that evaluation of commits (accept or abandon) is made sooner.

A hybrid structure integrates more but smaller individual commits, yielding larger feature contributions in total. Figure 6.7a indicates that the size of abandoned and accepted commits of the hybrid structure are both smaller than for branching (Figure 6.7a). Figure 6.7b shows the distribution of churn size of contributions. We can see that toggling has a high median value (with a median value of 390 lines of code) than branching (median of 165). A following t-test proves the significant difference. This is because toggling enables developers to integrate their work into code repository as soon as any change has been made, not having to wait until the whole features are completed. At the same time, it helps reduce integration conflicts and speed up testing, so that a feature could be finished and merged faster.

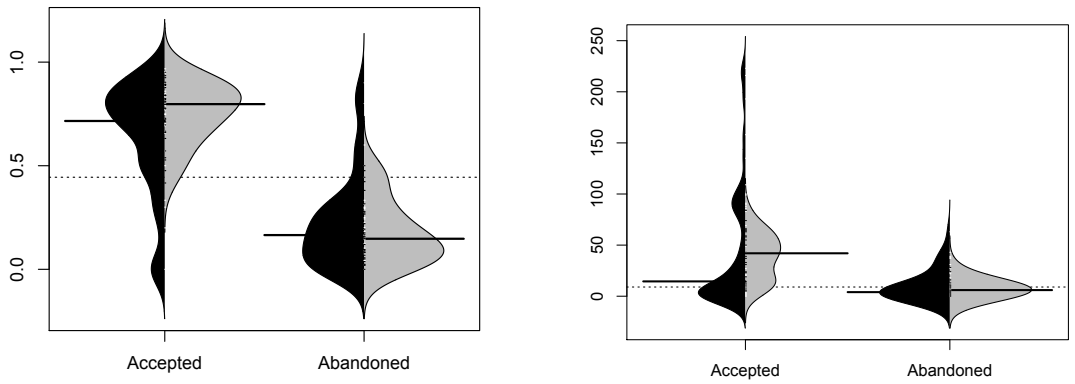
Hybrid and branching requires the same number of revisions to integrate commits. Figure 6.8 demonstrates the number of reviews a commit needs to go through before being accepted or abandoned. We find that the numbers do not differ between toggling and branching, and most of the commits are accepted or abandoned in the second round (have at least one patch version). This no difference is likely because integration does not impact the review process, in the sense that independently from integration mechanism developers still make mistakes in their initial patch submissions.

Hybrid accepts more and abandons less commits per calendar week. It also integrates more but smaller individual commits, yielding a large total contribution.



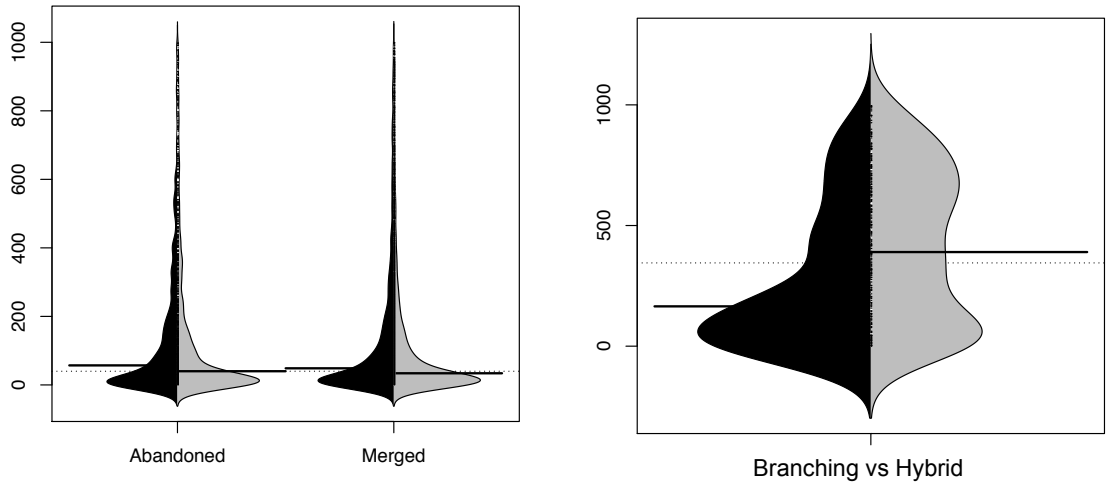
(a) Percentage of abandonment/acceptance (b) Number of abandonment/acceptance

Figure 6.5 Percentage/number of abandonment and acceptance per calender week (black: branching, grey: hybrid).



(a) Percentage of abandonment/acceptance on master branch (b) Number of abandonment/acceptance on master branch

Figure 6.6 Percentage/number of abandonment and acceptance on master branch per calendar week (black: branching, grey: hybrid).



(a) Size of abandonment/acceptance.

(b) LOC of contribution churn.

Figure 6.7 Characteristics of abandonment/acceptance (black: branching, grey: hybrid).

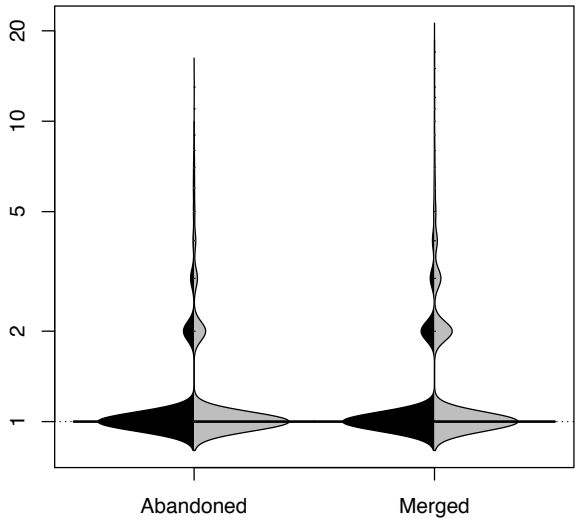


Figure 6.8 # patch revisions one commit goes through before being accepted/rejected.

Table 6.2 The median number/ratio of abandonment/merge of branching/hybrid per calendar week.

	branching	hybrid
sum of all patches per week	54.0	720.5
# abandoned patches	9.00	104.5
# merged patches	48.00	590.0
ratio of abandoned patches	19%	16%
ratio of merged patches	78%	82%

6.5 Threats to Validity

6.5.1 External Validity

In this work of comparing two integration techniques (branching and hybrid-toggling), we studied a commercial project from an industrial company, which adopts a hybrid integration approach combining branching and toggling. On one hand, this helps us reduce noise and focus on the same project. On the other hand, this restricts our analysis to a closed environment and cannot generalize our findings to projects in other different conditions such as open source systems.

However, access to software systems using toggles is still scarce [132], let alone projects that migrated for toggling. Hence, to our knowledge, our work is the first study on this topic.

6.5.2 Internal Validity

This project was launched as of May 2014 and started with branch-based integration for about six months, then migrated gradually to toggling approach. We studied one year of history of this project, assuming the first six months are branch-based and the latter six months are toggling-based. This short time period may not reveal the full characteristics of hybrid structure and impact our findings.

6.5.3 Construct Validity

The threat to construct validity concerns about the KPIs. The KPIs in this study are used by this company to measure the performance of individual developers and development activities. They are mainly proposed by managers based on their specific expectations and the reality of this company. Although this means that the company's developers will focus on these KPIs, since they are evaluated based on them, in a different company or context, there might be different key indicators such that the KPIs should be adjusted accordingly.

6.6 Conclusion and Recommendation

Branching and Toggling are two integration approaches. Branching isolates different teams by working on different feature branches, which causes costly integration conflicts. Toggling enables teams working together on different features on one branch, but with extra code complexity and technical debts.

Toggling has been adopted by many companies in practice like Google and Facebook. However, so far there is little quantitative work studying whether this technique really works better than branching. Companies wanting to try out this new technique lack a good guidebook to follow, while it might be too risky to migrate to this new integration approach completely.

In this paper, we conducted an empirical study to analyze a hybrid structure that combines both branching and toggling integration to see whether it works better than pure branching from two perspectives: "Integration effort" and "Productivity".

We find that this hybrid structure takes significantly less relative integration effort than branching, and makes larger contributions. In terms of productivity, a hybrid structure helps accept more commits and abandons less per calendar week, reduce waste and help integrate more production code.

Based on our case study results, we would recommend any company that wants to start adopting toggling for integration to:

In the beginning, it would be a good choice to start with a combination of branching and toggling. If the company wonders if there is potential risk of thoroughly migrating to a new structure, especially for a project that has been developed for a while, it then can try partially adopting toggling into the organization, such as the case study company. One can keep the old structure (e.g., branching-based), while using toggles can help A/B testing and introduce new features. This can also help save integration effort and improve productivity.

6.7 Acknowledgement

We would like to thank the cooperation of this anonymous company and the support from NSERC.

CHAPTER 7 ARTICLE 4: CO-EVOLUTION OF INFRASTRUCTURE AND SOURCE CODE - AN EMPIRICAL STUDY

Yujuan Jiang, Bram Adams

Abstract

Infrastructure-as-code automates the process of configuring and setting up the environment (e.g., servers, VMs and databases) in which a software system will be tested and/or deployed, through textual specification files in a language like Puppet or Chef. Since the environment is instantiated automatically by the infrastructure languages' tools, no manual intervention is necessary apart from maintaining the infrastructure specification files. The amount of work involved with such maintenance, as well as the size and complexity of infrastructure specification files, have not yet been studied empirically. Through an empirical study of the version control system of 265 OpenStack projects, we find that infrastructure files are large and churn frequently, which could indicate a potential of introducing bugs. Furthermore, we found that the infrastructure code files are coupled tightly with the other files in a project, especially test files, which implies that testers often need to change infrastructure specifications when making changes to the test framework and tests.[1]

7.1 Introduction

Infrastructure-as-code (IaC) is a practice to specify and automate the environment in which a software system will be tested and/or deployed [97]. For example, instead of having to manually configure the virtual machine on which a system should be deployed with the right versions of all required libraries, one just needs to specify the requirements for the VM once, after which tools automatically apply this specification to generate the VM image. Apart from automation, the fact that the environment is specified explicitly means that the same environment will be deployed everywhere, ruling out inconsistencies.

The suffix "as-code" in IaC refers to the fact that the specification files for this infrastructure are developed in a kind of programming language, like regular source code, and hence can be (and are) versioned in a version control system. Puppet [154] and Chef [150] are two of the most popular infrastructure languages. They are both designed to manage deployments

[1] This paper has been published in the conference of Mining Software Repositories 2015.

on servers, cloud environments and/or virtual machines, and can be customized via plug-ins to adapt to one's own working environment. Both feature a domain-specific language syntax that even non-programmers can understand.

The fact that IaC requires a new kind of source code files to be developed and maintained in parallel to source code and test code, rings some alarm bells. Indeed, in some respects IaC plays a similar role as the build system, which consists of scripts in a special programming language such as GNU Make or Ant that specify how to compile and package the source code. McIntosh et al. [114] have shown how build system files have a high relative churn (i.e., amount of code change) and have a high coupling with source code and test files, which means that developers and testers need to perform a certain effort to maintain the build system files as the code and tests evolve. Based on these findings, we conjecture that IaC could run similar risks and generate similar maintenance overhead as regular build scripts.

In order to validate this conjecture, we perform an empirical case study on 265 OpenStack projects. OpenStack is an ecosystem of projects implementing a cloud platform, which requires substantial IaC to support deployment and tests on virtual machines. The study replicates the analysis of McIntosh et al. [114], this time to study the co-evolution relationship between the IaC files and the other categories of files in a project, i.e., source code, test code, and build scripts. To get a better idea of the size and change frequency of IaC code, we first address the following three preliminary questions.

PQ1) *How many infrastructure files does a project have?*

> Projects with multiple IaC files have more IaC files than build files (median of 11.11% of their files). Furthermore, the size of infrastructure files is in the same ballpark as that of production and test files, and larger than build files.

PQ2) *How many infrastructure files change per month?*

> 28% of the infrastructure files in the projects changed per month, which is as frequently as production files, and significantly more than build and test files.

PQ3) *How large are infrastructure system changes?*

> The churn of infrastructure files is comparable to build files and significantly different with the other file categories. Furthermore, the infrastructure files have the highest churn per file (MCF) value among the four file categories.

Based on the preliminary analysis results, we then address the following research questions:

RQ1) *How tight is the coupling between infrastructure code and other kinds of code?*

> Although less commits change infrastructure files than the other file categories, the changes to IaC files are tightly coupled with changes to Test and Production files. Furthermore, the most common reasons for coupling between infrastructure and test are "Integration" of new test cases and "Updates" of test configuration in infrastructure files.

RQ2) *Who changes infrastructure code?*

> Developers changing the infrastructure take up the lowest proportion among all developers, and they are not dedicated to IaC alone, since they also work on production and test files.

7.2 Background and Related Work

7.2.1 Infrastructure as Code

IaC (Infrastructure as Code) makes it possible to manage the configuration of the environment on which a system needs to be deployed via specifications similar to source code. A dedicated programming language allows to specify the environment such as required libraries or servers, or the amount of RAM memory or CPU speed for a VM. The resulting files can be versioned and changed like any other kind of source code. This practice turns the tedious manual procedure of spinning up a new virtual environment or updating a new version of the environment (from the low-level operating system installed all the way up to the concrete application stack) into a simple click of executing a script. This automation and simplication helps shorten the release and test cycle, and reduces the potential of human error.

Currently, Puppet and Chef are the two most popular infrastructure languages. They allow to define a series of environment parameters and to provide deployment configuration. They can define functions and classes, and allow users to customize their own Ruby plug-ins according to their specific requirements. Figure 8.1 shows two code snippets of Puppet and Chef that realize the same functionality, i.e., initializing an https server on two different platforms (each platform requires a different version of the server).

```
# Chef snippet
case node[:platform]
when "ubuntu"
    package "httpd-v1" do
        version "2.4.12"
        action: install
    end
when "centOS"
    package "httpd-v2" do
        version "2.2.29"
        action: install
    end
end
```

```
# Puppet snippet
case $platform{
    'ubuntu': {
        package {'httpd-v1':
            ensure => "2.4.12"
        }
    }
    'centOS': {
        package {'httpd-v2':
            ensure => "2.2.29"
        }
    }
}
```

Figure 7.1 Code snippets of Puppet and Chef.

7.2.2 Related Work

Our work replicates the work of McIntosh et al. [114] who empirically studied the build system in large open source projects and found that the build system is coupled tightly with the source code and test files. In their work, they classify files into three different categories including "Build", "Production", and "Test". They also studied the ownership of build files to look into who spent the most effort maintaining these files. They observed different build ownership patterns in different projects: in Linux and Git a small team of build engineers maintain most of the build maintenance, while in Jazz most developers contribute code to the build system. In our work, we added a fourth category of files (IaC) and we focus the analysis on those files. Hindle et al. [93] studied the release patterns in four open source systems in terms of the evolution in source code, tests, build files and documentation. They found that each individual project has consistent internal release patterns on its own.

Similar to McIntosh et al. [114], other researchers also used "association rules" to detect co-change and co-evolution. Herzig et al. [91] used association rules to predict test failure and their model shows good performance with precision of 0.85 to 0.90 on average. Zaidman et al. [111] explored the co-evolution between test and production code. They found different coupling patterns in projects with different development style. In particular, in test-driven

projects there is a strong coupling between production and test code, while other projects have a weaker coupling between testing and development. Gall et al. [79] studied the co-evolution relation among different modules in a project, while our work focuses on the relation among different file categories.

Adams et al. [31] studied the evolution of the build system of the Linux kernel at the level of releases, and found that the build system co-evolved with the source code in terms of complexity. McIntosh et al. [113] studied the ANT build system evolution. Zanetti et al. [143] studied the co-evolution of GitHub projects and Eclipse from the socio-technical structure point of view. Our work is the first to study the co-evolution process between infrastructure and source code in a project.

Rahman et al. [131], Weyuker et al. [157], Meneely et al. [116] studied the impact of the number of code contributors on the software quality. Karus et al. [102] proposed a model that combines both code metrics and social organization metrics to predict the yearly cumulative code churn. It turns out that the combined model is better than the model only adopting code metrics. Bird et al. [61] proposed an approach for analyzing the impact of branch structure on the software quality. Nagappan et al. [125] conducted a study about the influence of organizational structure on software quality. Nagappan et al. [124] predicted defect density with a set of code churn measures such as the percentage of churned files. They showed that churn measures are good indicators of defect density.

Shridhar et al. [147] conducted a qualitative analysis to study the build file ownership styles, and they found that certain build changes (such as “Corrective” and “New Functionality”) can introduce higher churn and are more invasive. Our work is the first to study the ownership style of infrastructure files.

Curtis et al. [67], Robillard [137], Mockus et al. [121] [119] focus on how domain knowledge impacts the software quality. It turns out that the more experienced and familiar in a domain, the fewer bugs are introduced. This suggests that if IaC experts change IaC files, they are less likely to introduce bugs than, say, build developers. We study IaC ownership, yet do not study bug reports to validate this link.

To summarize, our study is the first to analyze the co-evolution of IaC with known file categories, as well as ownership of IaC file changes.

7.3 Approach

The process of our study is shown as a flow chart in Figure 8.4.

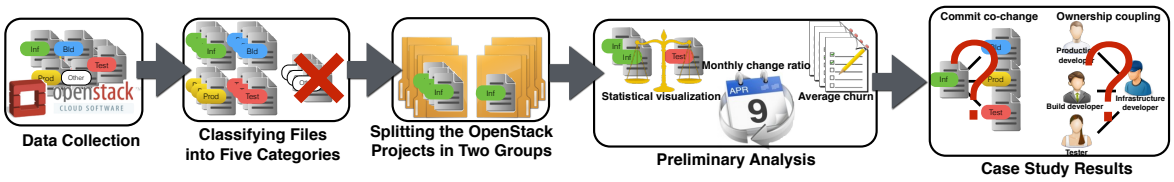


Figure 7.2 Flow chart of the whole approach.

7.3.1 Data Collection

OpenStack is an open source project launched jointly by Rackspace Hosting and NASA in July 2010. It is in fact governed by a consortium of organizations who have developed multiple interrelated components that together make up a cloud computing software platform that offers "Infrastructure As A Service (IaaS)". Users can deploy their own operating system or applications on OpenStack to virtualize resources like cloud computing, storage and networking.

Given that testing and deploying a cloud computing platform like OpenStack requires continuous configuration and deployment of virtual machines, OpenStack makes substantial use of IaC, adopting both "Puppet" and "Chef" to automate infrastructure management. Apart from its adoption of IaC, OpenStack has many other characteristics that prompted us to study it in our empirical study. In particular, it has 13 large components ("modules") spread across 382 projects with their own git repositories, 20 million lines of codes, rapid release cycle (cycle time of 6 months), and a wide variety of users such as AT& T, Intel, SUSE, PayPal, and eBay.

7.3.2 Classifying Files into Five Categories

In order to study the co-evolution relationship between the infrastructure and the other files, first we need to classify them into different categories. We mainly identify *"Infrastructure"*, *"Build"*, *"Production (i.e., source code)"* and *"Test"* files. Other files, such as images, text, and data files, were categorized into the *"Others"* category and were not considered in our study. Note that we classified any file that ever existed in an OpenStack project across all git commits, amounting to 133,215 files in total.

In order to do the classification, we used a similar approach as McIntosh et al. [114]. First, we wrote a script to identify files with known naming patterns. For example, names containing "test" or "unitest" should be test files, while the "Makefile" or "Rakefile" names are build files, and the files with a programming language suffix such as ".py", ".rb" (typically, but not always) belong to the production files. The infrastructure files written in Puppet have a suffix ".pp". After classifying those files that are easily identified, we manually classified the

remaining 25,000 unclassified files.

For this, we manually looked inside the files to check for known constructs or syntax, and for naming conventions specific to OpenStack. If the content was not related to any of the four categories of interest, we classified it as "Other". We put the resulting file classification online [2].

7.3.3 Splitting the OpenStack Projects in Two Groups

We collected all the revisions from the 382 git repositories of the OpenStack ecosystem. After classification, we did a statistical analysis and found 117 projects without infrastructure file, so we removed them from our data set, as we focus on the analysis of IaC. Upon manual analysis of the projects without infrastructure files, we found that they mostly do not serve the main functionality of OpenStack (namely cloud computing), but rather provide the supporting services like reviewing management (e.g., "reviewday"), test drivers (e.g., "cloudroast" and "cookbook-openstack-integration-test"), plugins for build support (e.g, "zmq-event-publisher"), front-end user interface generation (e.g., "bugdaystats" and "clif"), and external libraries (e.g., "cl-openstack-client" for Lisp).

Of the projects containing infrastructure files, we found that some of them have only few infrastructure files, while other projects have many infrastructure files. The former repositories are relatively small in terms of the number of files (with a median value of 71.5 and mean of 174.4). In order to study if the infrastructure system acts differently in these different contexts, we split the remaining projects into two groups: the "Multi" group with projects that contain more than one infrastructure file and the "Single" group with projects that contain only one infrastructure file. The Multi group contains 155 projects, whereas the Single group 110 projects. In our study, we compare both groups to understand whether there is any difference in maintenance effort between the two groups.

7.3.4 Association Rules

To analyze the coupling relationship in RQ1 and RQ2, we use association rules. An association rule is a possible coupling between two different phenomena. For example, supermarkets found that diapers and beer are usually associated together, because normally it is the father who comes to buy diapers and along the way may buy some beer for himself.

To measure the importance of an association rule, a number of important metrics can be calculated, such as **"Support"** (Supp) and **"Confidence"** (Conf). The metric Support(X)

[2] https://github.com/yujuanjiang/OpenStackClassificationList.

indicates the frequency of appearance of X, while the metric Confidence(X=>Y) indicates how often a change of X will happen together with a change of Y. For example, if there are 20 commits in total for project P, and 10 of them are changing infrastructure files, then Supp(Inf)=0.5 (10/20). If among these 10 commits, 6 of them also change test files, then Conf(Inf=>Build)=0.6 (6/10). Association rules can help us understand the coupling relationship among different projects. Note that we don't mine for new association rules, but analyze the strength of the six rules involving IaC files and the three other file categories (IaC =>Build, IaC =>Production, IaC =>Test and the three inverse rules).

Additionally, in the qualitative analysis of the change coupling, we need to select the most tightly coupled commits for manual analysis. Since Confidence is not a symmetrical measure (Conf(X=>Y) is different from Conf(Y=>X)), it yields two numbers for a particular pair of file categories, which makes it hard to identify the projects with "most tightly coupled" commits. For this reason, we adopt the metric "Lift", which measures the degree to which the coupling between two file categories is different from the situation where they would be independent from each other. For each project, we computed the Lift value, then for the 10 projects with the highest lift value for a pair of file categories, we selected the top 10 most tightly coupled commits. The formula for "Lift" related to the Conf and Supp metrics is as follows:

$$Conf = \frac{P(A \bigcap B)}{P(A)}$$

$$Supp = P(A \bigcap B)$$

$$Lift = \frac{P(A \bigcap B)}{P(A)P(B)}$$

7.3.5 Card Sorting

For our qualitative analysis, we adopted "Card Sorting" [117] [43], which is an approach that allows to systematically derive structured information from qualitative data. It consists of three steps: 1) First, we selected the 10 projects with the highest lift value for a certain pair of file categories. Then, for each such project, we wrote a script to retrieve all commits changing files of both categories. 2) Then, we randomly sort all selected commits for a project and pick 10 sample commits. 3) The first author then manually looked into the change log message and code of each commit to understand why this co-change happens. In particular, we were interested in understanding the reason why changes of both categories were necessary in the commit. If this is a new reason, we added it as a new "card" in our "card list". Otherwise, we just increased the count of the existing card. After multiple iterations, we obtained a list of reasons for co-change between two file categories. Finally, we clustered cards that had

related reasons into one group, yielding seven groups.

"Card sorting" is an approach commonly used in empirical software engineering when qualitative analysis and taxonomies are needed. Bacchelli et al. [40] used this approach for analyzing code review comments. Hemmati et al. [88] adopted card sorting to analyze survey discussions [147].

7.3.6 Statistical tests and beanplot vsualization.

In our work, we mainly used the Kruskal-Wallis and Mann-Whitney tests to do the statistical tests, and used the beanplot package in R as the visualization tool for our results.

The Kruskal-Wallis test [12] is a non-parametric method that we use to test if there exists any difference between the distribution of a metric across the four file categories. If the null hypothesis ("there is no significant difference between the mean of the four categories") is rejected, at least one of the categories has a different distribution of the metric under study. To find out which of the metrics has a different distribution, we then use Mann-Whitney tests as post-hoc test. We perform such a test between each pair of file categories, using the Bonferroni correction for the alpha value (which is 0.05 by default in all our tests).

A Beanplot [101] is a visualization of a distribution based on boxplots, but adding information about the density of each value in the distribution. Hence, apart from seeing major moments like median, minimum or maximum, one can also see which values are the most frequent in the sample under study. By plotting two or more beanplots next to each other, one can easily see asymmetry in the distribution of values (see Figure 7.5).

7.4 Preliminary Analysis

Before addressing the main research questions, we first study the characteristics of IaC files themselves.

PQ1: How many infrastructure files does a project have?

Motivation: As infrastructure code is a relatively new concept, not much is known about its characteristics. How common are such files, and how large are they in comparison to other known kinds of files?

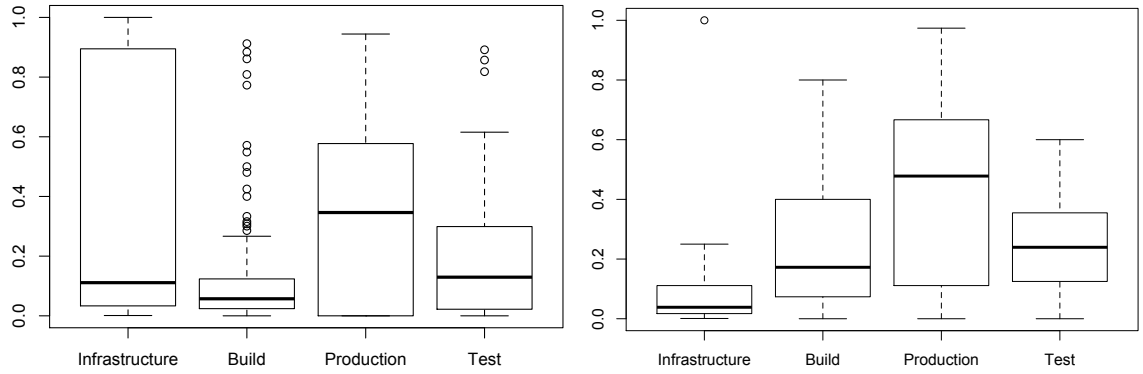
Approach: First, we compute the number and percentage of files in each file category for each project. Afterwards, we computed the number of lines of each infrastructure file in each project. Furthermore, we manually checked the Multi projects to understand why they contain such a high proportion of infrastructure files, for example whether they are

projects dedicated to IaC code. We also did Kruskal-Wallis and post-hoc tests to check the significance of the results with as null hypothesis "there is no significant difference among the distributions of the proportion/the LOC of each file category".

Results: Multi projects have a higher proportion of infrastructure files than build files, with a median value of 11.11% across projects. Figure 7.3 shows the boxplot of the proportion of the four file categories relative to all files of a project, while Table 7.1 shows the corresponding numbers. We can see that in both groups, the trends are the same except for infrastructure files. Unsurprisingly, the production files take up the largest proportion of files (with a median of 34.62% in group Multi and 47.80% in group Single). This makes sense, because the source code files should be the fundamental composition of projects. The test files take up the second largest proportion (with a median value of 12.95% in group Multi and 23.94% in group Single). By definition, for "Single" projects, the infrastructure files take up the lowest proportion (with a median of 3.85%), behind build files (with a median of 17.24%), while for Multi projects the order is swapped (with a median of 11.11% for Infrastructure and 5.71% for Build files). Indeed, Multi projects not only have more than one infrastructure file, they tend to have a substantial proportion of such files. The differences in proportion between IaC files and the three other categories all are statistically significant.

The percentage of infrastructure files has a large variance for "Multi" projects. The median value is rather small compared to the other three categories, but within four projects, the percentage of infrastructure files can reach as high as 100%. Therefore, we ranked all the projects by the percentage of infrastructure files (from high to low) and manually looked into the top ones. We found that those projects clearly are infrastructure-specific. 51 of these projects have names related to the infrastructure system (28 of them named after Puppet, 23 after Chef). For example, the "openstack-chef-repo" repository is an example project for deploying an OpenStack architecture using Chef, and the "puppetlabs-openstack" project is used to deploy the Puppet Labs Reference and Testing Deployment Module for OpenStack, as described in the project profile on GitHub [18] [25].

Although production and test files statistically significantly are larger, the size of infrastructure files is in the same ballpark. Figure 7.4 shows for each file category the boxplot across all projects of the median file size (in terms of LOC). We can see that the sizes of infrastructure (median of 2,486 for group Multi and 1,398 for group Single), production (median of 2,991 for group Multi and 2,215 for group Single) and test files (with a median of 2,768 for group Multi and 1,626 for group Single) have the same order of magnitude. The size of build files is the smallest (with a median of 54 for group Multi and 52 for group Single).



(a) groupMulti (b) groupSingle

Figure 7.3 Boxplot of median proportions of four file categories for each project across all projects (excluding other files)

Table 7.1 The proportion of the four file categories in each project (%) in terms of the number of files.

		Infrastructure	Build	Production	Test
Group Multi	1st Qu.	3.33	2.41	0.00	2.21
	Median	11.11	5.71	34.62	12.95
	Mean	38.40	11.72	31.84	18.04
	3rd Qu.	89.47	12.37	57.73	29.91
Group Single	1st Qu.	1.73	7.60	11.46	12.71
	Median	3.85	17.24	47.80	23.94
	Mean	8.02	25.41	42.25	24.26
	3rd Qu.	11.11	40.00	66.67	35.21

Infrastructure files take up a small portion of projects with a median value of 11.11%, but their size is larger than build files and in the same ballpark as code and test files.

PQ2: How many infrastructure files change per month?

Motivation: The results of PQ1 related to size and to some degree proportion of files could indicate a large amount of effort needed to develop and/or maintain the infrastructure files, both in Single and Multi projects. To measure this effort, this question focuses on the percentage of files in a category that are being changed per month [114]. The more files are touched, the more effort needed.

Approach: In order to see how often each file category changes, we computed the number of changed files per month of each project. To enable comparison across time, we normalize the number by dividing by the corresponding number of files in a category during that month,

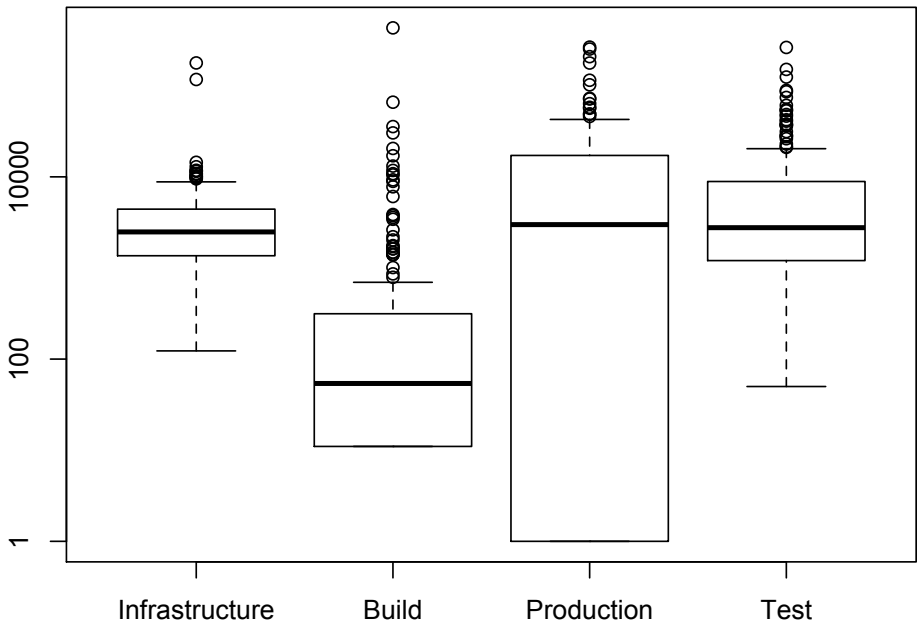


Figure 7.4 Boxplot of the median size (in LOC) of the four different file categories across all projects (group "Multi").

yielding the proportion of changed files. For each project, we then calculate the average proportion of changed files per month, and we study the distribution of this average across all projects. Note that we use average value in PQ2 and PQ3, whereas we use medians in the rest of the paper, because "Single" projects only have one infrastructure file and hence a median value would not make sense.

Furthermore, we again did a Kruskal-Wallis test and post-hoc tests to examine the statistical significance of our results.

Results: The average proportion of infrastructure files changed per month is comparable to that of source code, and much higher than that of build and test files, with a median value across projects of 0.28. Figure 7.5 shows the distribution across all projects in group "Multi" of the average proportion of changed files. Since the group "Single" has the same trend, we omitted its figure. We can see that the production files (with a median value of 0.28) change as frequently as the infrastructure files (with a median value of 0.28). The test files change less frequently (with a median value of 0.21) than infrastructure files but more frequently than build files (with a median value of 0.18).

The percentage of changed files per month for infrastructure and production files are significantly higher than for build and test files. A Kruskal-Wallis test on all file categories yielded a p-value of less than 2.2e-16, hence at least one of the file categories has a different distribution of monthly change percentage at 0.05 significance level. We then did

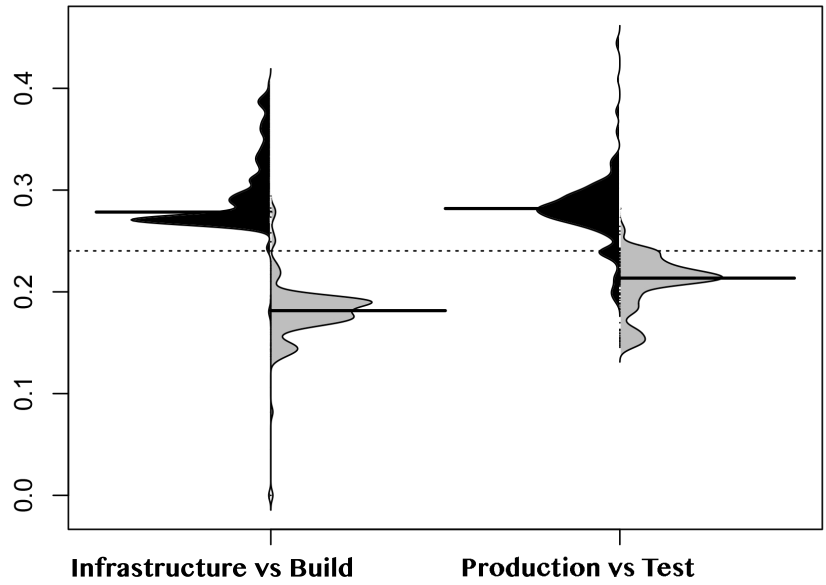


Figure 7.5 Distributions of average percentage of changed files per project for the four file categories (group "Multi").

Mann-Whitney post-hoc tests, which showed that, except for infrastructure and production files (p-value of 0.112 >0.05), the other file categories have a statistically significantly lower change proportion (p-value <0.05). Since change of source code is known to be an indicator of bug proneness [39] [105] or risk [145], infrastructure code hence risks to have similar issues.

> *The distribution of the median proportion of monthly change for infrastructure files is similar to production files, with a median value of 0.28, being higher than the build and test files.*

PQ3: How large are infrastructure system changes?

Motivation: Now that we know how many infrastructure files change per month for all the projects, we want to know how much each file changes as well. In addition to the number of changed lines, the types of changes matter as well. For example, two commits could both change one line of an infrastructure file, but one of them could only change the version

number while the other one may change a macro definition that could cause a chain reaction of changes to other files. Hence, we also need to do qualitative analysis on infrastructure files.

Approach: Churn, i.e., the number of changed lines of a commit is a universal indicator for the size of a change. In the textual logs of a git commit, a changed line always begins with a plus "+" or minus "-" sign. The lines with "+" are the newly added lines, while the lines with "-" are deleted lines from the code. In order to understand how much each file category changes per month, we define monthly churn of a project as the total number of changed lines (both added and deleted lines) per month. However, when we first looked into the monthly churn value, we found that there are many projects not active all the time, i.e., in certain months, the churn value is zero. Therefore, we just considered the "active period" for each project, namely the period from the first non-zero churn month until the last non-zero churn month.

To control for projects of different sizes, we also normalize the monthly churn of each project by dividing by the number of files of that project in each month. This yields the monthly churn per file (MCF). We study the distribution of the average value of churn and of the average value of MCF across all projects.

Results: The churn of infrastructure files is comparable to build files and significantly smaller than for production and test files. Figure 7.6 shows the beanplot of the monthly churn for both groups. We can see in group "Multi" that the test files have the highest average churn value (with a median value of 9), i.e., the test commits have the largest size, followed by production files (with a median value of 8). Infrastructure file changes (5.25) are larger than build file changes (4.25). In group "Single", the production files have the largest commits (median of 10), followed by test files and infrastructure files (both median of 8), and build files (median of 5). Kruskal-Wallis and post-hoc tests show that the distribution of churn of IaC files is significantly different from that of the other files categories, except from build files in the Single group.

The infrastructure files have the highest MCF value, with a median of 1.5 in group Multi and median of 5 in group Single. Figure 7.7 is the beanplot of the MCF for the two groups. We can see for both groups that the infrastructure files have the highest MCF value (median 1.5 in group Multi and median 5 in group Single), which means that the average change to a single infrastructure file is larger than for the other file categories. Hence, although the number of infrastructure files is smaller than the number of production files, there is proportionally more change being done to them.

The differences in average MCF between IaC files and the three other categories all are

statistically significant.

> *The churn of infrastructure files is comparable to that of build files, while the average churn per file of infrastructure is the highest across all file categories.*

7.5 Case Study Results

Now that we know that projects tend to have a higher proportion of Infrastructure files than build files, infrastructure files can be large, churn frequently and substantially, we turn to the main research questions regarding the coupling relation among commits and IaC ownership.

RQ1) *How tight is the coupling between infrastructure code and other kinds of code?*

Motivation: Based on the preliminary questions, we find that infrastructure files are large and see a lot of churn, which means that they might be bug prone. However, those results considered the evolution of each type of file separately from one another. As shown by McIntosh et al. [114], evolution of for example code files might require changes to build files to keep the project compilable. Similarly, one might expect that, in order to keep a project deployable or executable, changes to code or tests might require corresponding changes to the infrastructure code. This introduces additional effort on the people responsible for these changes. Kirbas et al. [106] conducted an empirical study about the effect of evolutionary coupling on software defects in a large financial legacy software system. They observed a positive correlation between evolutionary coupling and defect measures in the software evolution and maintenance phase. These results motivate us to analyze the coupling of infrastructure code with source, test and build code.

Approach: In order to identify the coupling relationship between changes of different file categories, we analyze for each pair <A, B>of file categories the percentage of commits changing at least one file of category A that also changed at least one file of category B. This percentage corresponds to the confidence of the association rule A=>B. For example, Conf(Infrastructure, Build) measures the percentage of commits changing infrastructure files that also change build files. Afterwards, we performed chi-square statistical tests to test whether the obtained confidence values are significant, or are not higher than expected due to chance.

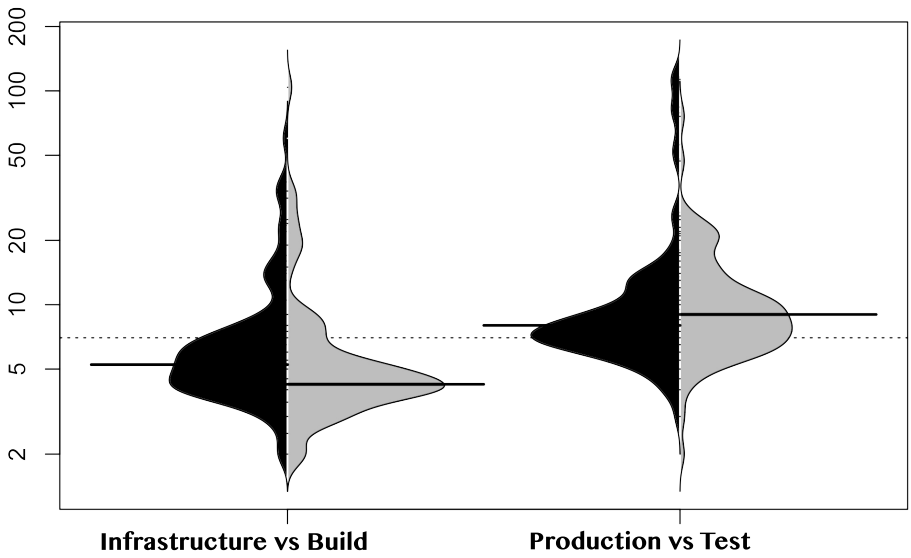


Figure 7.6 Beanplot of average monthly churn across all projects for the four file categories (group "Multi") (log scaled).

Finally, we also performed qualitative analysis of projects with high coupling to understand the rationale for such coupling. We used "card sorting" (see Section 7.3.5) for this analysis. We sampled 100 commits across the top 10 projects with the highest Lift metric (see Section 7.3.4) to look into why IaC changes were coupled so tightly with changes to other file categories.

Results: Infrastructure files change the least often in both groups. Table 7.2 shows the distribution of the Support and Confidence metrics across the projects in both groups while Figure 7.8 visualizes the distribution of these metrics for the group **Multi**. We do not show the beanplot of group **Single**, since it follows the same trends.

With group Multi as example, we can see that in terms of the Support metrics (i.e., the proportion of all commits involving a particular file category), source code changes occur the most frequently (with a median value of 0.3789), followed by test files (median of 0.2348), then build files (median of 0.1276). The infrastructure files change the least often, which suggests that the build and infrastructure files tend to be more stable than the other file categories. We observed the same behavior in group Single.

The commits changing infrastructure files also tend to change production and test files. In group Multi, Conf(Inf=>Prod) and Conf(Inf=>Test) have a high median value of 0.2637 and 0.4583 respectively. This indicates that most commits changing infrastructure files also change source code and test files. Conf(Inf =>Bld) has the lowest median value (0.0347), which indicates that commits changing infrastructure files don't need to change

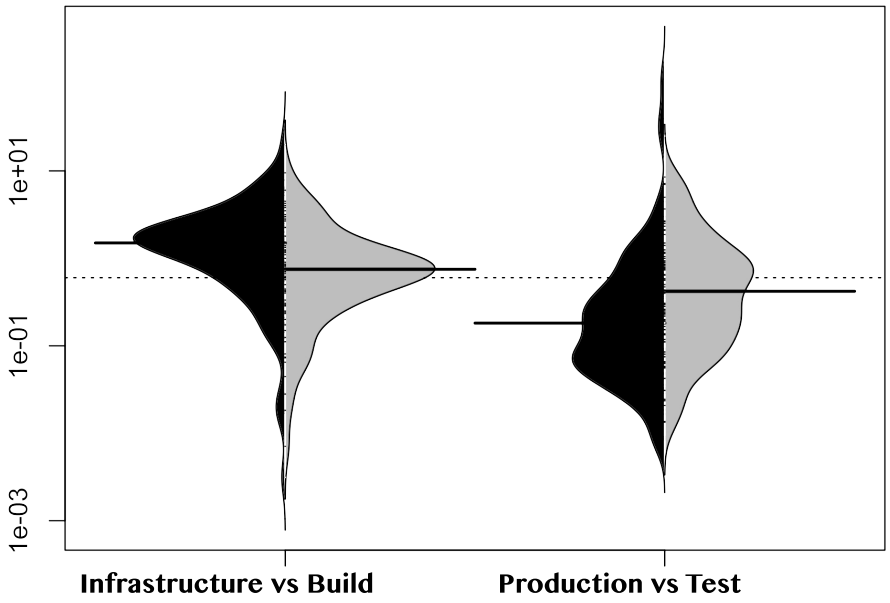


Figure 7.7 Beanplot of average MCF across all projects for the four file categories (group "Multi") (log scaled).

build files too often. Similar findings were made for group Single.

26% of test files require corresponding IaC changes in both groups. In group Multi, the Conf(Test =>Inf) metric has a higher value (median value of 0.2578) than Conf(Production =>Inf) (median value of 0.0885) and Conf(Build =>Inf) (median value of 0.1058). This means that one quarter of the commits changing test files also needs to change infrastructure files. Furthermore, infrastructure files also have a relatively high coupling with build files, i.e., around 11% of commits changing build files need to change infrastructure files as well. The observations in group Single follow the same trend.

The observed high confidence values are statistically significant in the majority of projects. Using a chi-square test on the confidence values, we found that in group **Multi**, among 155 projects, in 97 of them we observe a significant coupling between infrastructure and test files, and in 90 of them we observe a significant coupling between Infrastructure and Production files. In contrast, in only 2 of them we observe a significant coupling between Infrastructure and Build files. This means that the latter co-change relation statistically speaking is not unexpected, whereas the former ones are much stronger than would be expected by chance. In group "Single", among 110 projects, we found 33 that had a significant coupling between Infrastructure and Test files, and 35 of them that had a significant coupling between Infrastructure and production files, while there was no significant coupling observed between Infrastructure and Build files. Although the co-change relations with test

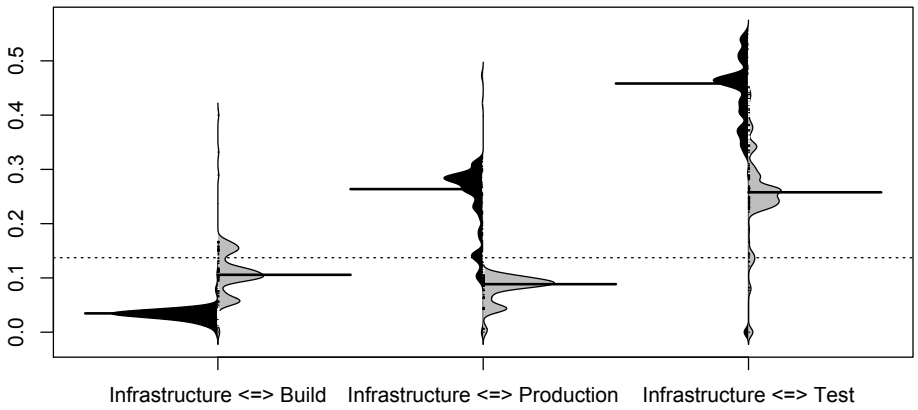


Figure 7.8 Distribution of confidence values for the coupling relations involving IaC files (Group Multi). The left side of a beanplot for A<=>B represents the confidence values for A =>B, while the right side of a beanplot corresponds to B =>A.

and production files are less strong than for the Multi group, they are still significant for many projects.

The most common reasons for the coupling between infrastructure and Build files are Refactoring and Update. Table 7.3 contains the resulting seven clusters from our card sort analysis. Those clusters group the different rationales that we identified by manually analyzing 100 commits. The table also contains for each cluster the percentage of the 100 commits that mapped to that cluster. Since each commit mapped to one cluster, the proportions add up to 100%.

The coupling between IaC and build files only had the third highest median Confidence. Coincidentally, the reasons for this coupling turn out to be simple, with three of the seven reasons absent. The most common reasons for this coupling include refactoring and updating files of both file categories because they share the same global parameters.

The most common reasons for the coupling between Infrastructure and Production files are External dependencies and Initialization. The most common reason for this coupling are changes in the IaC files to external dependencies like Ruby packages that require corresponding changes to the production files where these dependencies are used. Another common reason is initialization. If the project initializes a new instance of a client instance, it needs to initialize the parameters in the infrastructure file and add new source code for it.

The most common reasons for the coupling between infrastructure and test files are "Integration" and "Update". The coupling between infrastructure and test files has the highest value, and the reasons are spread across all seven reasons (similar to the coupling

Table 7.2 Median Support and confidence values for the coupling relations involving IaC files. Valued larger than 0.1 are shown in bold.

	system	Group Multi	Group Single
Support	Infrastructure	0.0402	0.0412
	Build	**0.1276**	**0.1324**
	Production	**0.3789**	**0.3806**
	Test	**0.2348**	**0.2344**
	Inf, Bld	0.0044	0.0044
	Inf, Prod	0.0336	0.0335
	Inf, Test	0.0585	0.0607
Conf	Inf =>Bld	0.0347	0.0343
	Inf = >Prod	**0.2637**	**0.2730**
	Inf =>Test	**0.4583**	**0.4673**
	Bld =>Inf	**0.1058**	**0.1140**
	Prod =>Inf	0.0885	0.0911
	Test =>Inf	**0.2578**	**0.2638**

between Infrastructure and Production). The most frequent reason is integrating new test modules into a project and updating the configuration for the testing process in the IaC files as well as in the test files.

Infrastructure files are changed less than the other file categories. The changes to Infrastructure files are tightly coupled with the changes to Test and Production files. The most common reasons for the coupling between Infrastructure and Test are "Integration" and "Update".

RQ2) *Who changes infrastructure code?*

Motivation: Herzig et al. [90] studied the impact of test ownership and team social structure on the testing effectiveness and reliability and found that they had a strong correlation. This means that in addition to the code-change relations of RQ1, we also need to check the relationship between the developers, i.e., the ownership among different file categories. Even though test or code changes might require IaC changes, it likely makes a difference whether a regular tester or developer makes such a change compared to an IaC expert.

Table 7.3 The reasons for high coupling and examples based on a sample of 300 (100*3) commits with confidence of 0.05.

Reason	IaC & Build		IaC & Production		IaC & Test	
Initialization	N/A	0	Initializing a new client instance needs to state copyright in both files.	**29%**	Integrating a new test specification or testing a bug fix in the infrastructure file. Initializing the value of the parameters of global methods in both the infrastructure and test file. Enabling a new test guideline (i.e., how tests should be performed) in new repositories (**most frequent**).	18%
External dependency	N/A	0	Importing new packages called in the source code, and defining their globally shared parameters.	**33%**	Adding a new function for testing external libraries or modules, then configuring the imported external libraries or other modules deployed by infrastructure (e.g., access to GitHub interface). Removing excessive dependencies.	12%
Textual Edit	Renaming, such as changing the project name from "Nailgun" to "FuelWeb".	2%	Re-formatting the text, or general cleanup like typos, quotes in global variables. Ensuring that a variable is changed consistently.	8%	Adding the license header for both file categories. Changing the value of shared method parameters such as the private variables of a test case.	3%
Refactoring	Getting rid of a Ruby gems mirror. Cleaning up the global variables such as the location directory. Removing code duplication.	**45%**	Removing old python schemas. Environment management overhaul. Changing coding standard and style.	7%	Refactoring or cleaning up the test configurations in the infrastructure files. Getting rid of a module everywhere.	14%
Organization of development structure	N/A	0	Separating a macro definition in an infrastructure file, then changing the data schema in source files correspondingly. Disabling a global functionality in infrastructure file.	10%	Switching the default test repository.	3%
Integration of new module	Adding RPM package specifications in an infrastructure file for use in the packaging stage of the build files.	9%	Integrating global bug fixes, or a new external module, such as JIRA.	3%	Enabling a new test module or integrating new test cases	**26%**
Update	Fixing errors in installation of packages. Ad-hoc change: changing Puppet manifest path in infrastructure file and build system bug fix in makefile. Changing installation configuration, such as installing from repos instead of mirrors, which requires removing the path and related parameters of the iso CD image in the infrastructure file and makefile.	**44%**	Updating external library dependencies, Changing global variables such as installation path. Defining a new function in infrastructure file, implemented in source code.	10%	Updating test configuration (e.g., changing default test repo location or default parameter value that is initialized in infrastructure files)	**24%**

Approach: First, we need to identify the ownership for each category. For this, we check the author of each commit. If the commit changes an infrastructure file, then we identify that commit's author as infrastructure developer. An author can have multiple identifications, e.g., one can be an infrastructure file and production developer at the same time (even for the same commit). We ignore those developers whose commits only change the files of the "Other" category.

We then compute the RQ1 metrics, but this time for the change ownership. Supp(Infrastructure) indicates the percentage of developers changing infrastructure files out of the total number of developers. Supp(Infrastructure, Build) is the percentage of developers changing both infrastructure and build files out of the total number of developers. Conf(Infrastructure, Build) is the percentage of IaC developers that also change build files out of the total number of developers changing at least once at infrastructure file.

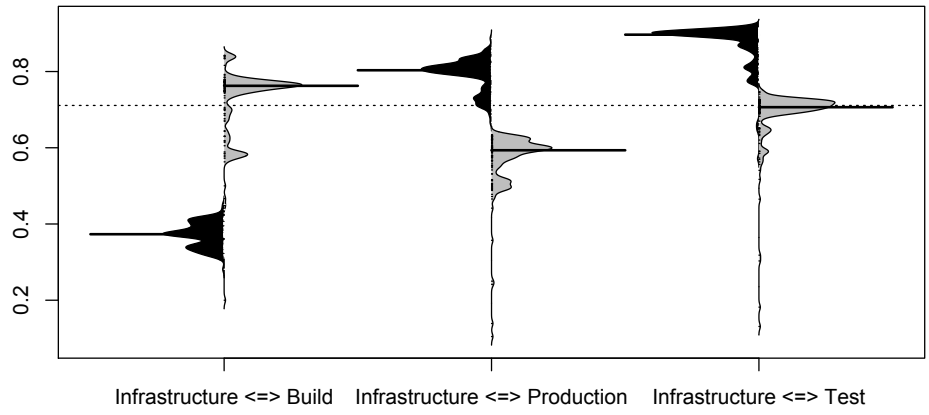


Figure 7.9 Distribution of confidence values for the coupling relations involving the owners of IaC files (Group Multi).

Result: Infrastructure developers have the lowest proportion among all developers while the production developers are the most common. Table 7.4 and Figure 7.9 show the measurements support and confidence metrics in terms of change ownership for the two groups of projects. Similar to RQ1, both groups have similar distribution trends, so we only show the beanplots for the Multi group.

We can see that, as expected, the developers of production code take up the highest proportion amongst all developers (with a median value of 0.6978), followed by test developers (median of 0.2744) and build developers (median of 0.5392). The infrastructure developers take up the lowest proportion (median of 0.2744).

The high value of metrics Supp(Inf, Prod) and Supp(Inf, Test) (with median values of 0.4442 and 0.4859 respectively) indicate that almost half of all developers in their career have had

Table 7.4 Median Support and confidence values for the coupling relations involving IaC developers. Values larger than 0.5 are shown in bold.

	system	Group Multi	Group Single
Support	Infrastructure	0.2744	0.2733
	Build	**0.5392**	**0.5384**
	Production	**0.7378**	**0.7390**
	Test	**0.6978**	**0.6977**
	Inf, Bld	0.2094	0.3726
	Inf, Prod	0.4442	0.4443
	Inf, Test	0.4859	0.4825
Conf	Inf =>Bld	0.3732	0.3726
	Inf = >Prod	**0.8034**	**0.8021**
	Inf =>Test	**0.8967**	**0.8962**
	Bld =>Inf	**0.7625**	**0.7635**
	Prod =>Inf	**0.5934**	**0.5940**
	Test =>Inf	**0.7064**	**0.7156**

to change at least one Infrastructure and Production file, or Infrastructure and Test file.

The majority of the infrastructure developers also develop Production or Test files. The Conf(Inf =>Prod) and Conf(Inf =>Test) both have a high value (with median of 0.8034 and 0.8967 respectively). This shows that most of the infrastructure developers are also production and test developers. In contrast, the metric Supp(Inf =>Bld) has the lowest value (median of 0.2094), which indicates that the developers changing infrastructure and build files respectively do not overlap substantially.

All kinds of developers change the IaC files. The Conf(Bld =>Inf), Conf(Prod =>Inf) and Conf(Test =>Inf) all have a very high value (median higher than 0.5). This shows that most of the build, production and test developers are infrastructure developers at the same time.

In particular, Conf(Inf =>Bld) has a lower value (median of 0.3732) compared to Conf(Bld =>Inf) (median of 0.7625). Build developers can be infrastructure developers, but the majority of infrastructure developers hardly change the build system, since so many other kinds of developers change the IaC files.

There is significant coupling between the Infrastructure and Test, and Infrastructure and Production ownership in most of the projects. In group **Multi**, for 107 projects we see that the coupling between Infrastructure and Test developers is signifi-

cant, in 76 projects the coupling between Infrastructure and Production, and in 20 projects the coupling between Infrastructure and Build change ownership. In group "Single", in 59 projects we see significant coupling between Infrastructure and Test developers, in 41 projects between Infrastructure and Production, and in 3 projects between Infrastructure and Build.

> *The infrastructure developers take up the lowest proportion among all developers. Developers working on infrastructure files are normal developers that also work on production and test files.*

7.6 Threats To Validity

Construct validity threats concern the relation between theory and observation. First of all, we use the confidence of association rules to measure how closely file categories and owners co-change in our research questions, similar to earlier work [114]. Furthermore, we use the monthly churn per file to measure the frequency of change and the number of changed lines of code to measure the amount of change. However, these metrics might not 100% reflect the actual coupling relationship and churn rate. Other metrics should be used to replicate our study and compare findings.

Threats to internal validity concern alternative explanations of our findings where noise could be introduced. During the classification of different file categories, we adopted the semi-automatic approach of McIntosh et al. [114], consisting of a script to separate certain file categories, then manually classifying the remaining files. To mitigate bias, at first the first author of this paper did this classification, followed by an independent verification by the second author.

Threats to external validity concern the ability to generalize our results. Since we have only studied one large ecosystem of open source systems, it is difficult to generalize our findings to other open and closed source projects. However, because OpenStack consists of multiple projects, and also has adopted the two most popular infrastructure tools Puppet and Chef, it is a representative case study to analyze. Further studies should consider other open and closed source systems.

7.7 Conclusion

IaC (Infrastructure as Code) helps automate the process of configuring the environment in which the software product will be deployed. The basic idea is to treat the configuration files as source code files in a dedicated programming language, managed under version control. Ideally, this practice helps simplify the configuration behavior, shorten the release cycle, and reduce the possible inconsistencies introduced by manual work, however the amount of maintenance required for this new kind of source code file is unclear.

We empirically studied this maintenance in the context of the OpenStack project, which is a large-scale open source project providing a cloud platform. We studied 265 data repositories and found that the proportion of infrastructure files in each project varies from 3.85% to 11.11%, and their size is larger than for build files and the same order of magnitude of code and test files (median value of 2,486 in group "Multi" and 1,398 for groups "Single"). Furthermore, 28% of the infrastructure files are changed monthly, significantly more than build and test files. The average size of changes to infrastructure files is comparable to build files, with a median value of 5.25 in group Multi and 8 in group Single (in terms of LOC). In other words, although they are a relatively small group of files, they are quite large and change relatively frequently.

Furthermore, we found that the changes to infrastructure files are tightly coupled to changes to the test files, especially because of "Integration" of new test cases and "Update" of test configuration in the infrastructure file. Finally, infrastructure files are usually changed by regular developers instead of infrastructure experts.

Taking all these findings together, we believe that IaC files should be considered as source code files not just because of the use of a programming language, but also because their characteristics and maintenance needs show the same symptoms as build and source code files. Hence, more work is necessary to study bug-proneness of IaC files as well as help reduce the maintenance effort.

Note that our findings do not diminish the value of IaC files, since they provide an explicit specification of a software system's environment that can automatically and consistently be deployed. Instead, we show that, due to their relation with the actual source code, care is needed when maintaining IaC files.

CHAPTER 8 ARTICLE 5: DOES INFRASTRUCTURE-AS-CODE EVOLVE AS CODE? - EMPIRICAL STUDY ON OPENSTACK AND MEDIAWIKI

Yujuan Jiang, Bram Adams

Abstract

Infrastructure-as-code (IaC) automates the process of configuring and setting up the environment (e.g., servers, VMs and databases) in which a software system will be tested and/or deployed, through textual specification files in a language like Puppet or Chef. Since the environment is instantiated automatically by the infrastructure language's tools, no manual intervention is necessary apart from maintaining the infrastructure specification files and the data that needs to be put inside the generated environment. The amount of work involved with such maintenance, as well as the size and complexity of infrastructure specification files, have not yet been studied empirically. Through an empirical study of the version control system of 50 OpenStack projects and 48 MediaWiki projects, we find that infrastructure-heavy projects contain up to 50% infrastructure-related files (median of 9 and 60 for MediaWiki and OpenStack respectively) with 11.8% and 25% of these infrastructure files of MediaWiki and OpenStack changing every month respectively. Furthermore, Infrastructure files tend to have a larger relative churn than production files, and they are coupled strongly to either Infrastructure-related data (MediaWiki) or test (OpenStack) files. These findings indicate a non-neligible amount of effort involved with maintaining IaC files.[1]

8.1 Introduction

Infrastructure-as-code (IaC) is a recent practice to specify and automate the environment (infrastructure) in which a software system will be tested and/or deployed [97]. For example, instead of having to manually configure the virtual machine on which a system should be deployed with a specific Linux distribution, web and database server, and the right versions of all required libraries installed, one just needs to specify the requirements for the VM once, after which tools automatically test and apply this specification to generate the VM image. Run-time support files such as web server configuration files can then be copied into the image to make it ready for deployment of the actual application under development. Apart

[1]This paper was invited to a special issue of journal Empirical Software Engineering, and got a major revision.

from automation, the fact that the environment is specified explicitly means that the same environment will be deployed everywhere, ruling out inconsistencies.

The recent surge in interest in the release engineering and DevOps [32] [46] [97] process have led to huge, with large investments by many big IT companies such as Google, Facebook and Mozilla. with the aim to release high quality software products faster to the end user. To help scale their deployment and release activities, these companies rely heavily on automation, in particular to instantiate the test and production environments, embracing IaC technologies.

While, initially, infrastructure had to be installed and managed manually, the advent of cloud, virtualization and containerization technologies has enabled the use of tools to automate infrastructure management. Although general scripting languages like Bash or Python could be used for this, dedicated IaC languages have emerged. Going back to at least 1993, cfengine was introduced in academia [20], enabling to configure the state of Unix-like systems with descriptive language. The surging interest in DevOps and release engineering, has sparked development of newer languages like Puppet (2005) and Chef (2009), followed more recently by Salt (2011) and Ansible (2012). Each of these languages can manage deployments on servers, cloud environments and/or virtual machines, and can be customized via plug-ins to adapt to one's own working environment. They typically feature a domain-specific language syntax that even non-programmers can understand. Similar to regular source code, they can be (and are) versioned in a version control system.

The fact that IaC requires a new kind of source code files to be developed and maintained in parallel to source code and test code could lead to new challenges for software maintenance. Indeed, in some respects, IaC plays a similar role as the build system, which consists of scripts in a special programming language such as GNU Make or Ant that specify how to compile and package the source code. McIntosh et al. [114] have shown how such build system files have a high relative churn (i.e., amount of code change) and have a high coupling with source code and test files, which means that developers and testers need to perform a certain amount of effort to maintain the build system files as the code and tests evolve. Based on these findings, we conjecture that IaC could run similar risks and generate similar maintenance overhead as regular build scripts.

In order to validate this conjecture, i.e., to better understand the potential maintenance effort of IaC code, we perform an empirical case study on 50 OpenStack projects and 48 MediaWiki projects. OpenStack is an ecosystem of projects implementing a cloud platform, which requires substantial IaC to support continuous integration and testing during development, while MediaWiki is a web-based wiki platform that forms the basis of many websites such as Wikipedia, and uses IaC to provide a consistent development environment to developers as

well as to deploy to production. The study replicates the analysis of McIntosh et al. [114], this time to study how IaC files and related files evolve compared to the other categories of files in a project, i.e., source code, test code, and build scripts:

RQ1) *How many infrastructure files does a project have?*

The top infrastructure projects of MediaWiki and OpenStack have a median of 9 and 60 IaC files, which yields a higher proportion up to 50% than the other file categories, except for production files. Furthermore, these projects have a median of 5.5 and 11 IaC-related data files, and a median of 0 and 2 test cases for IaC code.

RQ2) *How many infrastructure files change per month?*

A median of 11.8% and 25% of the infrastructure files of MediaWiki and OpenStack projects change every month. For OpenStack, this change rate is higher than all other file categories including Production files. We also found IaC test files have a higher change rate than IaC data files for MediaWiki, while for OpenStack we found the opposite.

RQ3) *How large are infrastructure file changes?*

Infrastructure files see a larger churn than Production files for MediaWiki, with a median value lines of code per month of 20.1 compared to 19.56 of Production files, while for OpenStack, infrastructure files have a smaller churn size (median of 21.97), compared to that of production files (31.96). When controlling for the number of IaC and production code files, infrastructure files have a higher relative churn than production files, with a median value lines of code per month of 2.03 and 0.240 respectively, compared to 0.02 and 0.024 of Production files.

RQ4) *How tight is the coupling between infrastructure-related files compared to coupling between traditional files?*

14.3% of the commits changing IaC code in MediaWiki change IaC-related data files, while for OpenStack 11.9% of the commits changing IaC code change IaC-related test files. Both types of coupling tend to be higher than the coupling of Production and Build files for MediaWiki and OpenStack, and even higher than coupling of Production and Test files for MediaWiki. Based on qualitative analysis, the most popular reasons for this coupling are “Integration” of new components or services, as well as “Modification” of global environment configuration.

RQ5) *Who changes infrastructure code?*

> 17% (MediaWiki) and 10% (OpenStack) infrastructure developers change InfraData code as well, higher than coupling between Production and Build developers (median of 11% and 5% for MediaWiki and OpenStack respectively), but lower than coupling between Production and Test developers (median of 32% and 74% respectively).

While the reasons for coupling identified in this paper are similar as for coupling between non-IaC files, unfortunately development environments for IaC are not as advanced yet as those for regular development of code, which opens many areas of future work. This paper extends our MSR 2015 paper [99] by adding an additional case study (MediaWiki), and using a finer-grained classification of files with extra categories of "InfraData" and "InfraTest". Furthermore, instead of considering all projects of OpenStack and MediaWiki, we analyze the RQs for the top IaC repositories, then use the top production code repositories as baseline. Although this does not allow to measure coupling between IaC and production code files directly, as was done in the previous paper (few of the analyzed repositories have both), it does allow to isolate better the effects of IaC code versus production code. The rest of the paper is organized as follows. We first introduce the background and related work (Section 8.2) and our approach (Section 8.3), followed by our case study results (Section 8.4). We conclude with threats to validity (Section 8.5) and the conclusion (Section 8.6).

8.2 Background and Related Work

8.2.1 Infrastructure-as-code

IaC (Infrastructure-as-code) makes it possible to configure the environment on which a system needs to be deployed via specifications similar to source code. A dedicated programming language allows specification of the environment, such as required libraries or servers, or the amount of RAM memory or CPU speed for a VM. The resulting files can be versioned and changed like any other kind of source code. Similarly, once the environment is generated, and before deploying the application under development into it, one needs to populate it automatically with run-time configuration and data files.

This practice turns the tedious manual procedure of spinning up a new virtual environment or updating a new version of the environment (from the low-level operating system installed all the way up to the concrete application stack) into a simple execution of a script. This automation and simplification helps streamline the development, test and deploy cycle, and reduces the potential of human error.

Currently, Puppet and Chef are the two most popular infrastructure languages. They allow to define a series of environment parameters, configure nodes, and specify which services or files should be deployed on each node. These languages provide support for reuse, allowing the definition of functions and classes, and the development of plug-ins for to customization. Figure 8.1 shows two code snippets of Chef and Puppet that realize the same functionality, i.e., installing specific versions of a package on two different platforms.

```
# Chef snippet
case node[:platform]
when "ubuntu"
    package "httpd-v1" do
        version "2.4.12"
        action: install
    end
when "centOS"
    package "httpd-v2" do
        version "2.2.29"
        action: install
    end
end
```

```
# Puppet snippet
case $platform{
    'ubuntu': {
        package {'httpd-v1':
            ensure => "2.4.12"
        }
    }
    'centOS': {
        package {'httpd-v2':
            ensure => "2.2.29"
        }
    }
}
```

Figure 8.1 Code snippets of Puppet and Chef.

Apart from installing a server or generating a virtual image, infrastructure code typically also needs to copy infrastructure-related data such as configuration files, SSH keys or certificates into the generated environment, in order to support the environment. We refer to such data files as infrastructure data ("InfraData"). An example InfraData code snippet is shown in Figure 8.2. This snippet from MediaWiki contains configuration data for the nginx webserver, such as server status and port number. Note the port variable, which gets its value from the IaC code during instantiation of the infrastructure.

Finally, similar to regular source code, IaC code should be tested to make sure it executes as requested. Puppet and Chef both provide test frameworks for infrastructure code. Usually they are placed under a directory named "spec" or "rspec". We refer to such test files as infrastructure test files ("InfraTest"). An example InfraTest code snippet from MediaWiki is shown in Figure 8.3. This snippet is used to test if component "mesos" is version 0.14.

```
# Nginx status and server metrics site
server {
    listen      127.0.0.1:<%= @port %>;
    server_name  localhost;
    location    /nginx_status {
        stub_status  on;
        access_log   off;
        allow        127.0.0.1;
        deny         all;
    }
}
```

Figure 8.2 InfraData code snippet.

8.2.2 Related Work

IaC emerged as a best practice in the DevOps community [46] [97], aiming to help reduce manual work and its potential risk/effort, and bridge the gap between development and operation process. Spinellis [148] stated that IaC is an essential technology to reduce the pressure and complexity of installing systems manually, without considering the potential cost of IaC code. However, thus far, little work has been done to study IaC code. Delaet et al. [70] conducted a survey on 11 popular automation configuration tools (such as BCFG2, Cfengine 3, Puppet and Chef) from four different perspectives, including their specification properties (e.g., language type and modularization mechanisms), deployment properties (e.g., scalability and architecture), specification management properties (e.g., usability and versioning support), and support (e.g., available tutorial and community). Based on this survey, the authors proposed a framework for users to quickly figure out which tools fit their requirement the most. They also claimed in the paper that "Puppet has a lot of confusing terminology" and "Chef is hard to use because of its syntax and the use of a lot of custom terminology". Hummer et al. [98] proposed a test framework for Chef, in order to make sure a Chef recipe as a whole can make a system converge to a desired state. These works indicate that IaC code might need effort to maintain. However, to our knowledge no work has been done to quantitatively analyze the evolution process of IaC related code and compare it to that of

```
require 'spec_helper'
    describe 'mesos' do
        context 'with given version' do
            let(:version) { '0.14' }
            let(:params) {{:ensure => version}}

            it { should contain_package('mesos').with(
            { 'ensure' => version
            })}
        end
    end
```

Figure 8.3 InfraTest code snippet.

Production code.

Our work replicates the work of McIntosh et al. [114], who empirically studied the build system in large open source projects and found that the build system is coupled tightly with the source code and test files. In their work, they classify files into three different categories including "Build", "Production", and "Test", and also studied the ownership of build files to look into who spent the most effort maintaining these files. They observed different build ownership patterns in different projects: in Linux and Git a small team of build engineers maintains most of the build maintenance, while in Jazz most developers contribute code to the build system. In our MSR 2015 paper, we added a fourth category of files (IaC) and we focused the analysis on those files. In this extension, we revisit this study by adding another case study and focusing on the characteristics and coupling of infrastructure-related files, and comparing them to the corresponding results for non-infrastructure code.

We used "association rules" to measure coupling between two different phenomena. Similar to McIntosh et al. [114], other researchers also used "association rules" to detect co-change and co-evolution. Herzig et al. [91] used association rules to predict test failure and their model shows a good performance with precision of 0.85 to 0.90 on average. Zaidman et al. [111] explored the co-evolution between test and production code. They found different coupling patterns in projects with different development style. In particular, in test-driven projects there is a strong coupling between production and test code, while other projects have a weaker coupling between testing and development. Gall et al. [79] studied the co-evolution relation among different modules in a project, while our work focuses on the relation among

different file categories.

Hindle et al. [93] studied the release patterns in four open source systems in terms of the evolution in source code, tests, build files and documentation. They found that each individual project has consistent internal release patterns on its own. Gregorio et al. [138] conducted a case study on the KDE project, classifying all the artifacts into nine types and analyzing the relationship amongst. They found that beyond work on source code, a project could devote much effort to other tasks. This motivates our concern that other than production code, other artifacts in a project such ad IaC code can cost effort.

Adams et al. [31] studied the evolution of the build system of the Linux kernel at the level of releases, and found that the build system co-evolved with the source code in terms of complexity. McIntosh et al. [113] studied the ANT build system evolution. Zanetti et al. [143] studied the co-evolution of GitHub projects and Eclipse from the socio-technical structure point of view. Our MSR 2015 paper [99] is the first to study the co-evolution between infrastructure and source code in a project, and this extension is the first to compare the evolution process between infrastructure-related code and production-related code files.

Rahman et al. [131], Weyuker et al. [157], Meneely et al. [116] studied the impact of the number of code contributors on software quality. Karus et al. [102] proposed a model that combines both code metrics and social organization metrics to predict the yearly cumulative code churn. It turns out that the combined model is better than the model only adopting code metrics. Bird et al. [61] proposed an approach for analyzing the impact of branch structure on the software quality. Nagappan et al. [125] conducted a study about the influence of organizational structure on software quality. Nagappan et al. [124] predicted defect density with a set of code churn measures such as the percentage of churned files. They showed that churn measures are good indicators of defect density. This work motivates our third research question to explore the monthly churn for infrastructure-related files.

Shridhar et al. [147] conducted a qualitative analysis to study build file ownership styles, and they found that certain build changes (such as “Corrective” and “New Functionality”) can introduce higher churn and are more invasive. Our work (the MSR 2015 paper [99] and this extension) is the first to study the ownership style of infrastructure files. Curtis et al. [67], Robillard [137], Mockus et al. [119, 121] focus on how domain knowledge impacts the software quality. It turns out that the more experienced and familiar to a domain, the fewer bugs are introduced. This suggests that if IaC experts change IaC files, they are less likely to introduce bugs than, say, build developers. We study IaC ownership, yet do not study bug reports to validate this link.

To summarize, our study is the first to compare the evolution and ownership of IaC with

that of known file categories.

8.3 Approach

The methodology of our study basically consists of four steps as is shown as a flow chart in Figure 8.4. First, we collected all revisions from the git repositories of MediaWiki and OpenStack, then we classified them into six categories (and one "miscellaneous" category, which is ignored). Next, we selected the top repositories containing the most IaC code and Production code respectively. Finally, we compared the characteristics of both groups of repositories (e.g., composition of repository, change rate and monthly churn) as well as the coupling between IaC-related code and Production-related code in terms of code change and ownership.

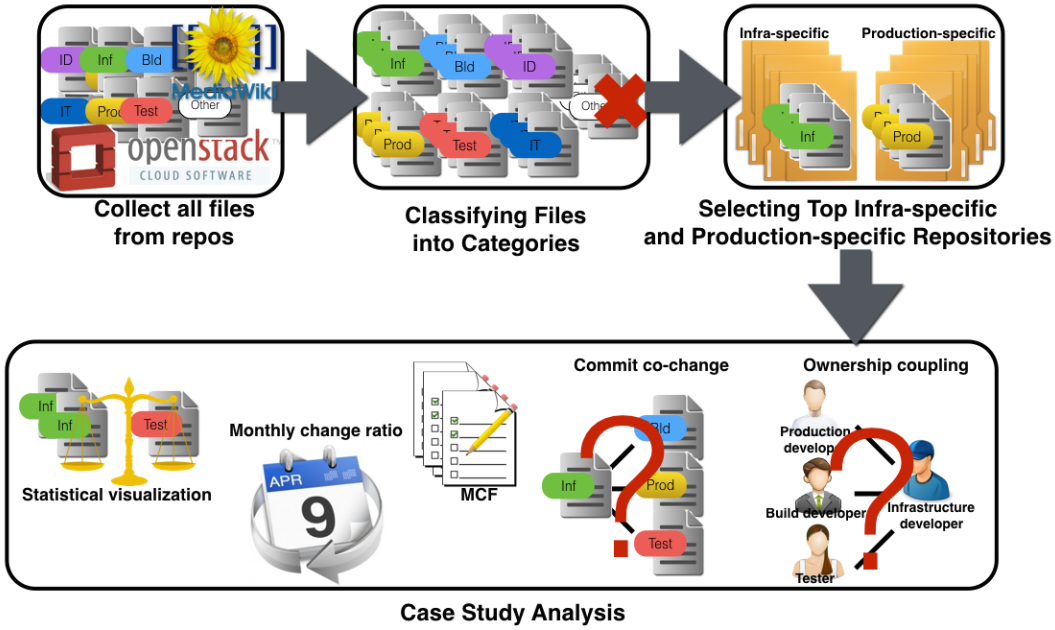


Figure 8.4 Flow chart of the whole approach.

8.3.1 Data Collection

We analyzed two large, popular, long-lived open source ecosystems, i.e., OpenStack and MediaWiki. Both have a large number of repositories and developers, while at the same time they are also following up-to-date release engineering practices, having invested substantial effort and resources into IaC. While OpenStack uses IaC to manage its continuous integration and testing processes, MediaWiki uses it during regular development as well as to

deploy production servers, Furthermore, MediaWiki projects often follow the best practices of Humble et al. [97] to separate build/test/source code from infrastructure code by using different repositories, while OpenStack instead stores its infrastructure-related code in its regular version control repositories. This difference makes these two systems interesting to compare to each other.

OpenStack

OpenStack [16] is an open source project launched jointly by Rackspace Hosting and NASA in July 2010. It is in fact governed by a consortium of organizations who have developed multiple interrelated components that together make up a cloud computing software platform that offers "Infrastructure As A Service (IaaS)". Users can deploy their own operating system or applications on OpenStack to virtualize resources like cloud computing, storage and networking.

Given that continuous integration and testing of a cloud computing platform like OpenStack requires continuous configuration and deployment of virtual machines, OpenStack makes substantial use of IaC, adopting both "Puppet" and "Chef" to automate infrastructure management. For the continuous integration and testing process of OpenStack, IaC code helps to create server nodes on demand and manage CI machines on each. Then IaC code will enable to automatically transfer/manage data such as passwords, SSH keys and so on.

Apart from its adoption of IaC, OpenStack has many other characteristics that prompted us to study it in our empirical study. In particular, it has 13 large components ("modules") spread across 382 projects with their own git repositories, 20 million lines in total, a release cycle time of 6 months, and a wide variety of users such as AT&T, Intel, SUSE, PayPal, and eBay.

MediaWiki

MediaWiki [22] is a free open source web platform written in PHP with a MySQL backend. It is the web platform of one of the most popular websites in the world, i.e., Wikipedia. MediaWiki is also internally adopted by many known companies and government, such as Intel and the United States Department of State. Its first official version was released in 2002, and by now there are already 1,480 projects totally. The MediaWiki organization started supporting IaC, i.e., Puppet and Chef, since 2011 to enable developers to work in a consistent development environment (e.g., using the MediaWiki-Vagrant portable IDE), or to manage deployment to production servers.

8.3.2 Classifying Files into Categories

In order to address our research questions, first we need to classify files into different categories. In our prior work, we classified all files into four categories: *"Infrastructure"*, *"Build"*, *"Production (i.e., source code)"* and *"Test"* files. In this extension, we add two more categories "InfraData (Infrastructure Data)" and "InfraTest (Infrastructure Test)". The "InfraData" files are data source files that are used by IaC, such as configuration files, images or databases, while the "InfraTest" files are test cases especially designed to test infrastructure code, using for example the rspec test framework [5] [24] [71]. Other unrelated files were categorized into the *"Others"* category and were not considered in our study. Note that we classified any file that ever existed in an OpenStack or MediaWiki project across all git commits amounting to 133,215 files for OpenStack and 241,698 for MediaWiki in total.

In order to do the classification, we used a similar approach as McIntosh et al. [114]. First, we wrote a script to identify files with known naming patterns. For example, names containing "test" or "unittest" should be test files, while the "Makefile" or "Rakefile" names are build files, and the files with a programming language suffix such as ".py", ".rb" for OpenStack and ".php" for MediaWiki typically (but not always) belong to the production files. The infrastructure files written in Puppet have a suffix ".pp" or can be found under a directory named "cookbook", "recipes" or "attributes". After classifying those files that are easily identified, we manually classified the remaining 25,000 unclassified files for OpenStack and 20,386 for MediaWiki. To do this, we manually looked inside the files to check for known constructs or syntax, and for naming conventions specific to OpenStack and MediaWiki. For example, we found that in general the test files for Infrastructure code are under a directory named spec or rspec. We put the resulting file classification online[2].

8.3.3 Selecting Top Infra-specific and Production-specific Repositories

In this paper, we would like to compare the way in which both case study systems maintain and take advantage of IaC (Infrastructure As Code) in practice. Given that both projects follow different practices, with MediaWiki often storing its IaC-related projects in separate repositories, while OpenStack also puts IaC files inside regular code repositories, we use a different analysis approach than was done in our MSR study of OpenStack.

Instead of dividing the Git projects of MediaWiki/OpenStack based on the number of IaC files that they have (none, one, or more), here we rank the projects based on the number of IaC files, then consider the topmost projects as "Infra-specific". Similarly, we also rank

[2]https://github.com/yujuanjiang/OpenStackClassificationList

the projects based on the number of production files, and consider the topmost projects as "Production-specific". We believe that the number of files is a better data selection criterion than the number of commits, since our goal is to study infrastructure-rich projects. Our study then compares the characteristics of the IaC code in the Infra-specific projects to those of production code in the Production-specific projects. Note that the "Production-specific" projects might as well contain little infrastructure code and vice versa, but no project ended up as Infra- and Production-specific at the same time.

OpenStack: We collected all the revisions from the 382 available git repositories of the OpenStack ecosystem. After classification and selection, we got 25 projects for both Infra-specific and Production-specific groups.

MediaWiki: We collected all the revisions from the 1,480 repositories from MediaWiki, then selected those with more than five infrastructure files. This resulted into 23 repositories. Finally, we selected the top 25 projects in terms of production code as Production-specific group. Eventually, we end up with 48 repositories in total, 23 of which are infrastructure-specific ("Infra-specific" group) and 25 are regular production code repositories ("Production-specific").

8.3.4 Statistical tests and beanplot visualization.

In our work, we mainly used the Kruskal-Wallis and Mann-Whitney tests to do statistical tests, and used the beanplot package in R as the visualization tool for our results.

The Kruskal-Wallis test [12] is a non-parametric method that we use to test if there exists any difference between the distribution of a metric across the six file categories. If the null hypothesis ("there is no significant difference between the mean of the candidate categories") is rejected, at least one of the categories has a different distribution of the metric under study. To find out which of the metrics has a different distribution, we then use Mann-Whitney tests as post-hoc test. We perform such a test between each pair of file categories, using the Bonferroni correction for the alpha value (which is 0.05 by default in all our tests).

A beanplot [101] is a visualization of a distribution based on boxplots, but adding information about the density of each value in the distribution. Hence, apart from seeing major moments like median, minimum or maximum, one can also see which values are the most frequent in the sample under study. By plotting two or more beanplots next to each other, one can easily see asymmetry in the distribution of values (e.g., Figure 8.7).

8.3.5 Change Coupling between File Categories

To analyze the coupling relationships in RQ4 and RQ5, we use association rules. An association rule is a possible coupling between two different phenomena. For example, McIntosh et al. [114] found that if a work item changes a source code file, there was a probability of as high as 27% that a build file had to change as well.

To measure the strength of an association rule (and hence coupling), a number of metrics can be calculated, such as **"Support"** (Supp) and **"Confidence"** (Conf):

$$Conf = \frac{P(A \bigcap B)}{P(A)}$$

$$Supp = P(A \bigcap B)$$

The metric Support(X) indicates the probability of appearance of X out of all items' appearance, while the metric Confidence(X=>Y) indicates the probability a change of X will happen together with a change of Y. For example, if there are 20 commits in total for project P, and 10 of them are changing infrastructure files, then Supp(Inf)=0.5 (10/20). If among these 10 commits, 6 of them also change InfraTest files, then Conf(Inf=>InfraTest)=0.6 (6/10). The confidence of an association rule can help us understand the coupling relationship among different file categories. Note that we do not mine for new association rules, but analyze the strength of the eight rules involving IaC/Production files and the two other file categories (IaC =>InfraData, IaC =>InfraTest, Production =>Build, Production =>Test, and the four inverse rules).

Additionally, in the qualitative analysis of the change coupling, we need to select the most tightly coupled commits for manual analysis. Since Confidence is not a symmetrical measure (Conf(X=>Y) is different from Conf(Y=>X)), it yields two numbers for a particular pair of file categories, which makes it hard to identify the projects with the "most tightly coupled" commits. For this reason, we adopt the metric "Lift", which measures the degree to which the coupling between two file categories is different from the situation where they would be independent from each other. For each project, we computed the Lift value, then for the 10 projects with the highest lift value for a pair of file categories, we selected the top 10 most tightly coupled commits. The formula for "Lift" related to the Conf and Supp metrics is as follows:

$$Lift = \frac{P(A \bigcap B)}{P(A)P(B)}$$

8.3.6 Qualitative Analysis using Card Sorting

For our qualitative analysis, we adopted "Card Sorting" [117, 43], which is an approach that allows to systematically derive structured information from qualitative data. It consists of three steps: 1) First, we selected the 10 projects with the highest lift value for a certain pair of file categories. Then, for each such project, we wrote a script to retrieve all commits changing files of both categories. 2) Then, we randomly sorted the retrieved commits for a project and picked 10 sample commits. 3) The first author then manually looked into the change log message and code of each commit to understand why this co-change happens. In particular, we were interested in understanding the reasons why changes of both categories were necessary in the commit. If this is a new reason, we added it as a new "card" in our "card list". Otherwise, we just increased the count of the existing card. After all cards are sorted, we obtained a list of reasons (about 10 to 15 reasons for different coupling) for co-change between two file categories. Finally, we clustered cards that had related reasons into one group, yielding seven groups eventually.

"Card sorting" is an approach commonly used in empirical software engineering when qualitative analysis and taxonomies are needed. Bacchelli et al. [40] used this approach for analyzing code review comments. Hemmati et al. [88] adopted card sorting to analyze survey discussions [147]. We performed card sorting on both MediaWiki and OpenStack data set.

8.4 Case Study Results

RQ1: How many infrastructure files does a project have?

Motivation: As infrastructure code is a relatively new concept, not much is known about its characteristics, for example "how common are such files?", and "how large are they in comparison to other known kinds of files?". Hence, our first research question helps to obtain a rough picture about the role of infrastructure files in a project.

Approach: First, we count the number and compute the percentage of files in each file category for each project. Afterwards, we computed the size of files in terms of number of lines of code, for each file category. Furthermore, we manually checked those projects with a large number of infrastructure files to understand why they contain such a high proportion of infrastructure files. We used Kruskal-Wallis and post-hoc tests to check the significance of the results with as null hypotheses "there is no significant difference among the distributions of the proportion (resp. LOC) of each file category".

Results: In Infra-specific projects, the top infrastructure projects have 9 (MediaWiki) and 60 (OpenStack) infrastructure files respectively, take up the largest

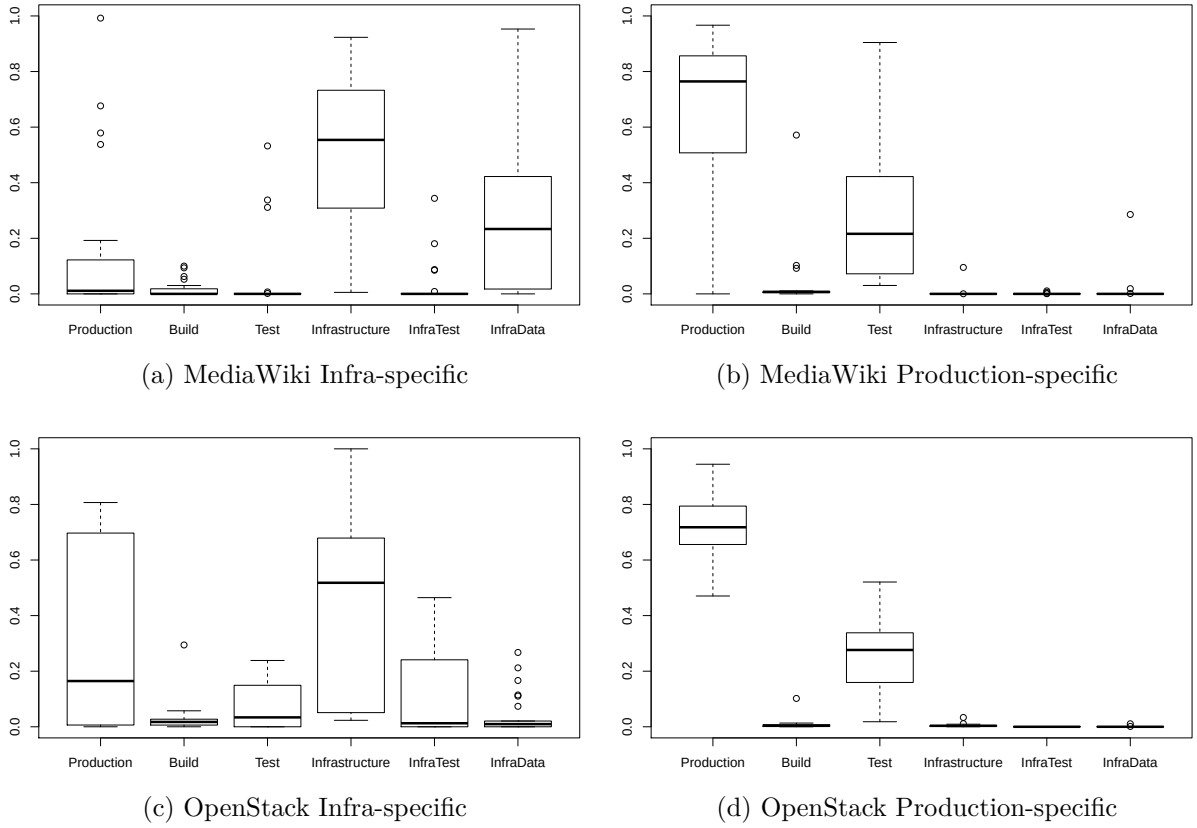
proportion up to 50%, while in Production-specific groups, the production code files dominate the project with a median value of 724.5 and 797 respectively. Figure 8.5 shows the boxplots of the proportion of the six file categories relative to all files of MediaWiki and OpenStack, while Table 8.1 shows the absolute number and Table 8.5 shows the corresponding proportions (see appendix) of files. In absolute numbers, this means that MediaWiki Infra-specific projects have a median of 9 IaC files compared to 724.5 production files in the Production-specific projects, compared to 60 IaC files and 797 production files for OpenStack. In Production-specific projects, Production files and Test files represent the majority of files, with only few other files. These results makes sense, because the source code files are the fundamental composition of projects. However, the proportion of infrastructure files in each system can reach up to 50%.

However, there exists a difference in these two projects in terms of the Infra-specific projects. For the Infra-specific group of MediaWiki, Production files represent only a median proportion of 1%. For OpenStack, the production files in the Infra-specific group still take up a high proportion (with a median value of 16%), right after Infrastructure code. InfraTest and InfraData files (with median value of 1% respectively), and as large as build files (1%). This mix of IaC and production code goes against best practices suggested by Humble et al. [**?**] to always separate IaC and regular production code.

MediaWiki Infra-specific projects proportionally have more InfraData files than InfraTest files, while OpenStack has the opposite. In terms of Infra-specific groups, MediaWiki has a larger proportion of InfraData files (median of 23.3%) while in OpenStack, InfraData has the lowest proportion. However, in terms of concrete numbers (as shown in Table 8.1), the number of InfraData files of OpenStack is actually higher than that of MediaWiki, with a median value of 11.00 compared to the 5.50 of MediaWiki. That said, IaC developers of OpenStack have discussed the need for more unit tests for their IaC before [23].

For both MediaWiki and OpenStack, the percentage of infrastructure files has a large variance for "Infra-specific" projects. To better understand the outliers in our data, we manually looked into the top projects with a large Infrastructure file proportion. We found that 11 of the OpenStack Infra-specific projects have names related to the infrastructure system (7 of them named after Puppet, 4 after Chef) and 15 projects in WikiMedia have a name containing "puppet". For example, the "openstack-chef-repo" repository is an example project for deploying an OpenStack architecture using Chef, and the "puppetlabs-openstack" project is used to deploy the Puppet Labs Reference and Testing Deployment Module for OpenStack, as described in the project profile on GitHub [18] [25].

The size of Infrastructure files is the largest across all file categories in Infra-



(a) MediaWiki Infra-specific

(b) MediaWiki Production-specific

(c) OpenStack Infra-specific

(d) OpenStack Production-specific

Figure 8.5 Proportion of number of files for Infra-specific and Production-specific projects respectively.

specific groups. Figures 8.6d are boxplots across all six file categories of file size (in terms of LOC) for the Infra-specific group of MediaWiki and OpenStack, while Table 8.6 shows the statistics. We can see that the size of Infrastructure files in Infra-specific group is larger than that of other file categories, with a median of 47 and 76 lines of code for MediaWiki and OpenStack respectively. For MediaWiki, the InfraData files (in the Infra-specific group) are as large as Production files, ranking second (median of 20) after Infrastructure files. For OpenStack's Infra-specific projects, InfraTest files and InfraData files (in Infra-specific group) are as large as Production and Test files, ranking second after Infrastructure and Build files.

Infrastructure files in Infra-specific projects are smaller than production code in Production-specific projects (and than test files, for OpenStack). As shown in Figure 8.6b, the Production files dominate the Production-specific projects, although for OpenStack, Test files are the largest (Figure 8.6d). The Infrastructure files in OpenStack's Production-specific group have a size as large as Build files (see Table 8.6).

Table 8.1 Distribution of the number of files for the MediaWiki and OpenStack projects respectively.

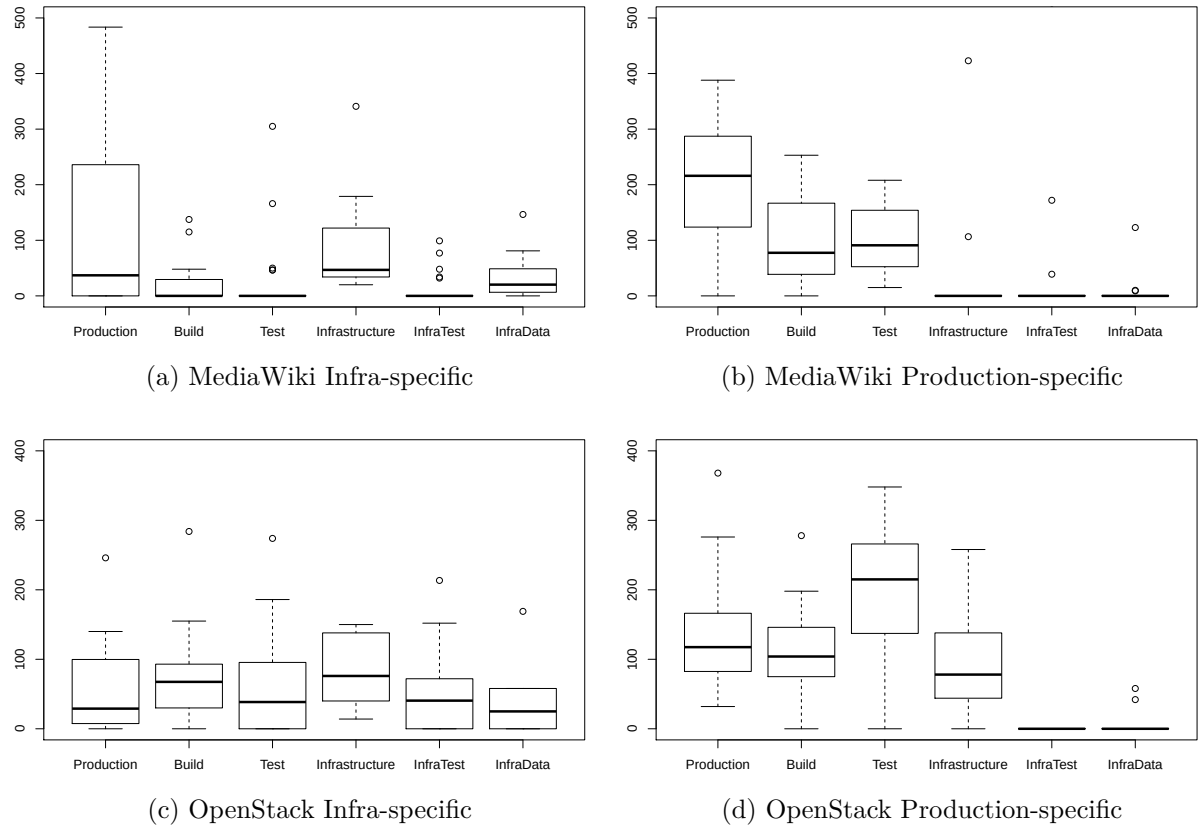
			Production	Build	Test	Infrastructure	InfraTest	InfraData
MediaWiki	Infra-specific	1st Qu.	0.0	0.00	0.0	6.0	0.00	1.75
		Median	1.0	0.00	0.0	9.0	0.00	5.50
		Mean	434.5	15.08	218.3	140.8	18.96	43.96
		3rd Qu.	12.0	1.00	0.0	52.0	1.00	26.50
	Production-specific	1st Qu.	389.0	4.5	118.2	0.0	0.0	0.0
		Median	724.5	9.0	283.5	0.0	0.0	0.0
		Mean	1994.4	43.0	6883.9	0.2	18.8	0.8
		3rd Qu.	2519.8	39.0	548.8	0.0	0.0	0.0
OpenStack	Infra-specific	1st Qu.	1	2.00	0.0	43.0	0	0.00
		Median	14	4.00	12.0	60.0	2	11.00
		Mean	351	26.04	100.9	275.4	62	62.96
		3rd Qu.	839	16.00	183.0	185.0	52	11.00
	Production-specific	1st Qu.	564	1.00	177.0	3.00	0	0
		Median	797	6.00	316.0	4.00	0	0
		Mean	1079	9.44	473.2	5.96	0.21	0.72
		3rd Qu.	1496	9.00	462.0	5.00	0	0

Infra-specific projects have a median number of 9 (MediaWiki) and 60 (OpenStack) IaC files, which is proportionally lower than the number of Production, but higher than Build and Test files of Production-specific projects. IaC files have a median size of 47 (MediaWiki) and 76 (OpenStack) lines of code, which is smaller than Production files.

RQ2: How many infrastructure files change per month?

Motivation: The results of RQ1 related to number and to some degree size of files could indicate a certain amount of effort needed to develop and/or maintain infrastructure files, in both projects. To measure this effort, this question focuses on the percentage of files in a category that are being changed per month [114], i.e., the "monthly churn". The more files are touched, the more maintenance effort would be needed.

Approach: In order to see how often each file category changes, we computed the number of changed files per month of each project. To enable comparison across time, we normalize the number by dividing by the corresponding number of files in a category existing that month, yielding the proportion of monthly changed files in each category. For each project and file



(a) MediaWiki Infra-specific

(b) MediaWiki Production-specific

(c) OpenStack Infra-specific

(d) OpenStack Production-specific

Figure 8.6 File size (LOC) for the two groups of MediaWiki and OpenStack projects' respectively.

category, we then calculate the median proportion of changed files per month. Finally, we compared each file category's distribution of this median across all projects.

Furthermore, we again did a Kruskal-Wallis test and post-hoc tests to examine the statistical significance of any differences in the average proportion of changed files between file categories.

Results: The Infrastructure files change more frequently than InfraTest and InfraData files, in both projects. Figure 8.7 shows the distribution of proportion of monthly changed file per project, while Table 8.7 in the appendix shows the statistics. Figures 8.7a and 8.7c show for Infra-specific groups that in both projects, more Infrastructure files (median of 0.118 and 0.250 for MediaWiki and OpenStack respectively) change per month than InfraTest and InfraData files (median of 0.036/0.019 and 0.115/0.134 respectively). However, in the MediaWiki Infra-specific group, slightly more InfraTest files change per month than InfraData files, while for OpenStack we can see the opposite. This might be because those file categories are not that common in MediaWiki/OpenStack, respectively (see RQ1), hence

it is easier to change a larger proportion of them.

In the Production-specific projects, Production files change more frequently than Build files. Figures 8.7b and 8.7d demonstrate the change for Production-specific groups. We can see that in both cases, Production files have a monthly change proportion of 0.277 (MediaWiki) and 0.229 (OpenStack), compared to 0.132 (MediaWiki) and 0.174 (OpenStack) for Build files. For MediaWiki, more Production files change than Test files as well (median of 0.123).

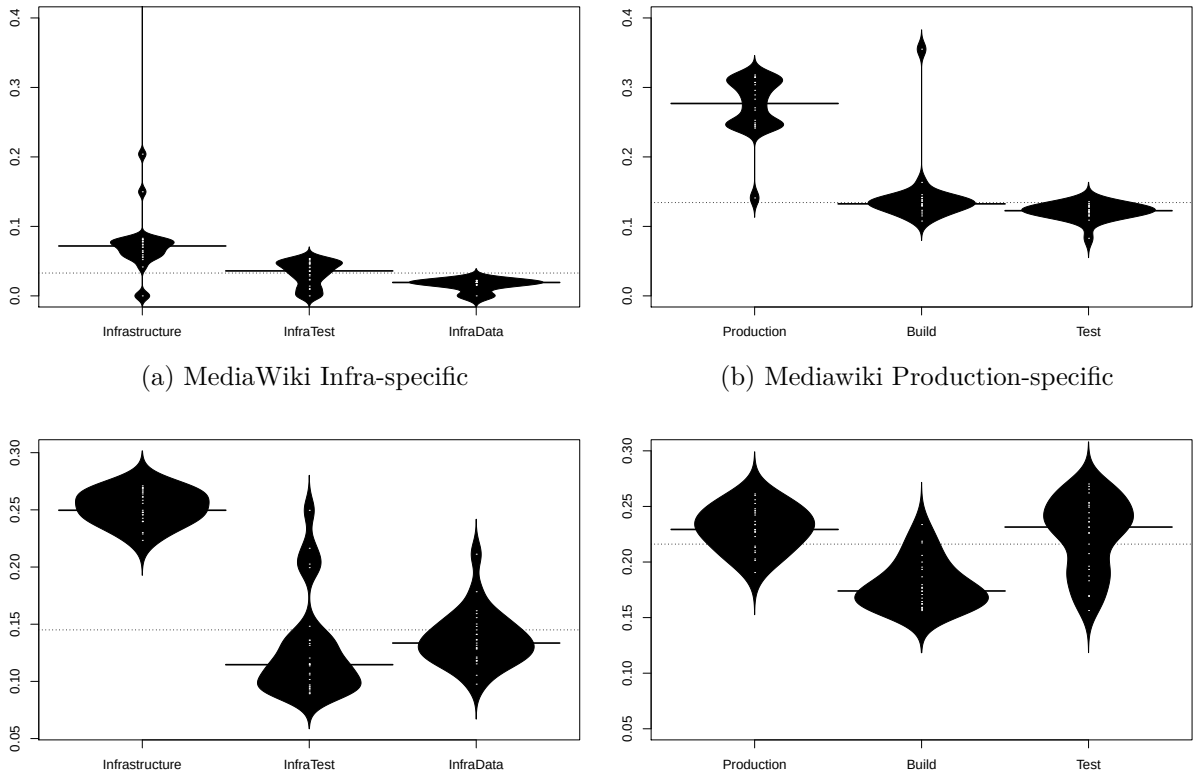
For MediaWiki, Production files in the Production-specific group change more frequently than Infrastructure files in Infra-specific group, while for OpenStack, Infrastructure files have the highest change rate. Figures 8.7a and 8.7b show how Production files have the highest change rate amongst all (median of 0.277), while Infrastructure files change less than Production, Build and Test files (median of 0.118 compared to 0.277, 0.132 and 0.123). On the other hand, Figure 8.7c and 8.7d show that Infrastructure code changes the most frequently every month (with a median value of 0.25), much more frequently than the Production (0.201), Build (0.228) and Test (0.163) code of Production-specific projects, and the InfraTest and InfraData files of Infra-specific projects. This confirms our earlier findings for OpenStack [99].

Infrastructure files change more frequently than InfraTest and InfraData files in Infra-specific groups of both MediaWiki and OpenStack, while they change even more frequently than Production files in the Production-specific OpenStack projects. However, in MediaWiki, they change less frequently than Production, Build and Test files.

RQ3: How large are infrastructure file changes?

Motivation: Now that we know how many infrastructure files change per month for all the projects, we want to know how much each file changes as well. For example, even though IaC files change more frequently than production files in OpenStack, they might only involve changes to one or two lines compared to typically larger code changes.

Approach: Churn, i.e., the number of changed lines of a commit, is a universal indicator for the size of a change. In the textual logs of a git commit, a changed line always begins



(a) MediaWiki Infra-specific

(b) Mediawiki Production-specific

(c) OpenStack Infra-specific

(d) OpenStack Production-specific

Figure 8.7 Proportion of changed files for Production-specific and Infra-specific projects, respectively.

with a plus "+" or minus "-" sign. The lines with "+" are the newly added lines, while the lines with "-" are deleted lines from the code. In order to understand how much each file category changes per month, we define its monthly churn of a project as the total number of changed lines (both added and deleted across all its files) per month.

To control for projects and file categories of different sizes, we also normalize the monthly churn of each file category by dividing by the number of files of that file category in each month. This yields the monthly churn per file (MCF). Finally, similar to RQ2, we calculate for each project and file category the median monthly churn and median MCF across all months, then compare the file categories' distribution of these median values across all projects.

Results: For MediaWiki, Infrastructure files have a larger monthly churn than InfraTest and InfraData files. Figures 8.8a and 8.8c show the distribution of monthly change rate of Infra-specific projects for MediaWiki and OpenStack. We can see that for MediaWiki, the Infrastructure files have the highest churn value (median of 20.01 lines of code) compared to InfraTest and InfraData (median 0.00 and 1.0). However, for the Open-

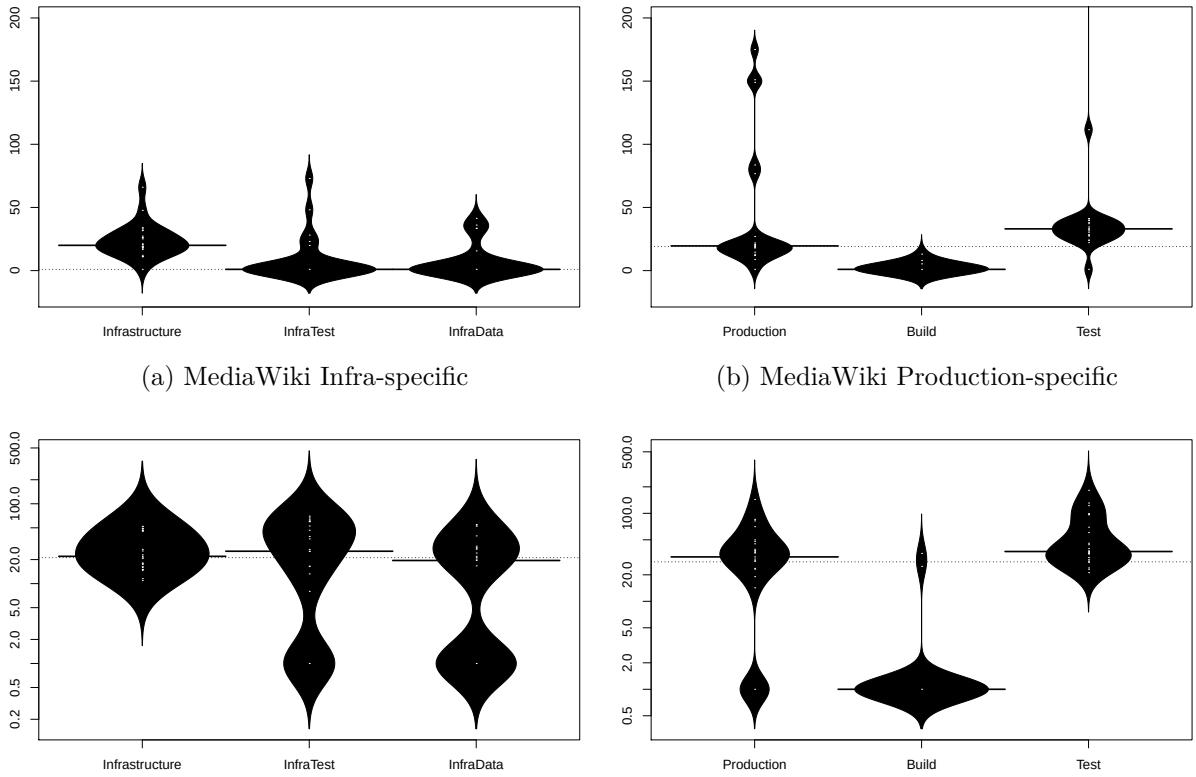
Stack Infra-specific group, Infrastructure files (median of 21.97) have a monthly churn size comparable to InfraTest (median of 25.45) and InfraData (median of 19.49) files.

Production files have a larger monthly churn than Build files (median of 19.56/31.96 vs 1.40/1.0 for MediaWiki and OpenStack respectively), while this churn is comparable to that of Test files in both MediaWiki and OpenStack projects. Figure 8.8b and 8.8d show the monthly churn distribution of Production-specific projects for MediaWiki and OpenStack, while Table 8.8 in the appendix shows the corresponding statistics. Production files have a monthly churn of 19.56 lines of code compared to 1.4 for Build files in MediaWiki, and a median of 31.96 compared to 1.0 for OpenStack respectively. Production files are almost as large as Test files, with a median of 33.4 and 36.81 lines of code respectively for MediaWiki and OpenStack.

For MediaWiki, the Infrastructure files of Infra-specific projects have a larger monthly churn value than Production files in the Production-specific group, while for OpenStack, we see opposite. Infrastructure files of Infra-specific group have a median value of monthly churn of 20.01, higher than 19.56 of Production files of Production-specific group. However, for OpenStack, the monthly churn size of Infrastructure files is smaller (median of 21.97 lines of code) than Production files (median of 31.96 lines of code).

For MediaWiki, Infrastructure files have larger relative churn (MCF) than InfraTest and InfraData files, while for OpenStack InfraTest files have a higher MCF value. Figures 8.9a and 8.9c show the MCF value for Infra-specific group of both MediaWiki and OpenStack. We can see that for MediaWiki, Infrastructure files have highest MCF value (median of 2.03 lines of code per file), followed by InfraTest (median of 1.00) and InfraData (median of 0.16), while for OpenStack, Infrastructure files have a lower value than InfraTest (median of 1.00), but higher than InfraData (median of 0.241).

For both MediaWiki and OpenStack, the Infrastructure files of Infra-specific projects have a higher MCF value than Production files in the Production-specific group. In order to control our churn findings for the impact of the number of files, we normalized the monthly churn of each file category by the corresponding number of files. Infrastructure files have a median MCF value of 2.03 and 0.240 lines of code per file for MediaWiki and OpenStack respectively compared to a median of 0.02 and 0.024 lines of code per file for Production files of Production-specific group. Additionally, Production files have a lower MCF value than Build and Test files of the Production-specific group.



(a) MediaWiki Infra-specific

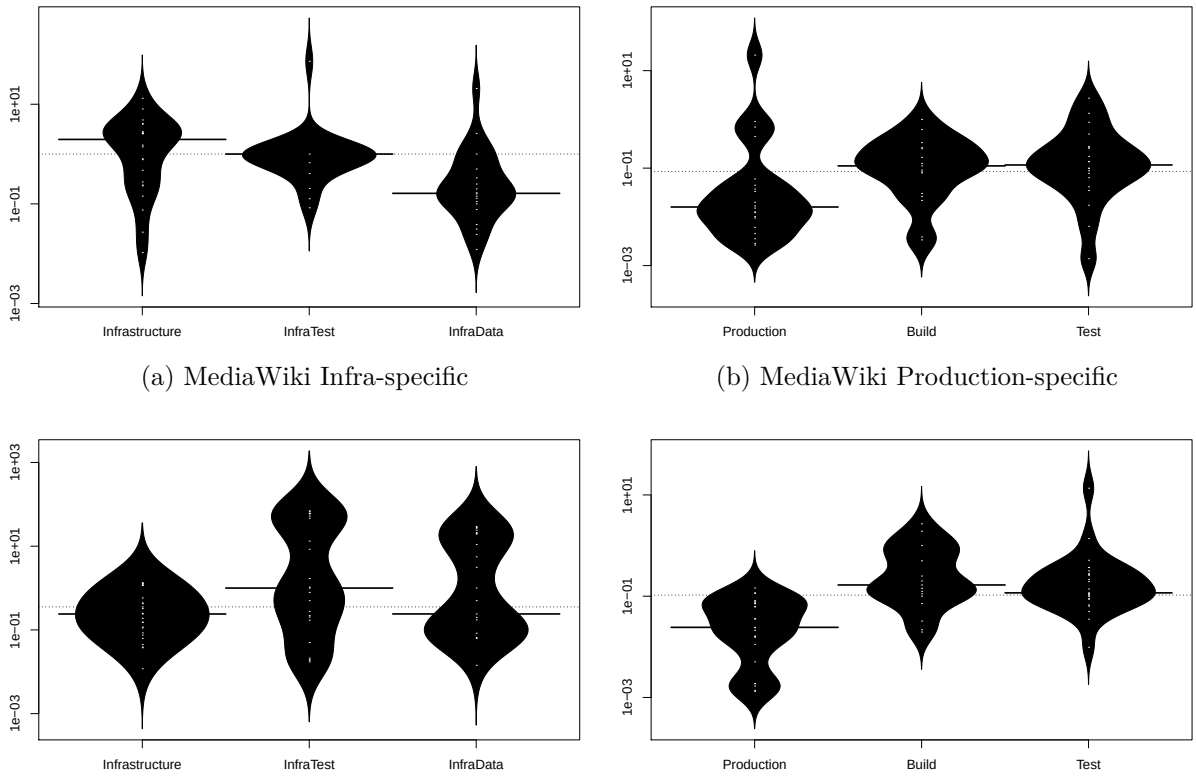
(b) MediaWiki Production-specific

(c) OpenStack Infra-specific (log-scaled)

(d) OpenStack Production-specific (log-scaled)

Figure 8.8 Churn size.

In the new project groups (Infra-specific and Production-specific), we found that Infrastructure files have a larger monthly churn than Production files for MediaWiki, but smaller for OpenStack. After being normalized, both MediaWiki and OpenStack ended up with Infrastructure files of larger relative churn (MCF) than Production files. This means that the already high churn for Infrastructure Files is concentrated even more in the lower number of files compared to Production files. This could indicate that Infrastructure files potentially are more defect-prone (higher churn in lower number of files), based on defect model research on regular source code.



(a) MediaWiki Infra-specific

(b) MediaWiki Production-specific

(c) OpenStack Infrastructure-specific

(d) OpenStack Production-specific

Figure 8.9 MCF value for MedianWiki and OpenStack (log-scaled).

RQ4: How tight is the coupling between infrastructure-related files compared to coupling between traditional files?

Motivation: Based on the results of prior research questions, we find that OpenStack's infrastructure files are large and see a lot of churn, which means that they might be bug prone (based on similar observations on production code [36] [37] [126] [133] [152]). However, the previous research questions considered the evolution of each type of file separately from one another. As shown by McIntosh et al. [114], evolution of, for example, code files might require changes to build files to keep the project compilable. Similarly, one might expect that, in order to keep a project deployable or executable, changes to IaC code might require corresponding changes to the infrastructure data and/or tests. This introduces additional effort on the people responsible for these changes. Kirbas et al. [106] observed a positive correlation between evolutionary coupling and defect measures in an empirical study about the effect of evolutionary coupling on software defects in a large financial legacy software system. Since, contrary to our previous work [99], this study considers repositories that either excel in IaC code or Production code, but not both (as this rarely happens in MediaWiki),

we do not study direct coupling between IaC and production code. Instead, we calculate coupling between Infrastructure and InfraTest/InfraData files in Infra-specific projects, and compare this coupling to the normal coupling between Production, Build and Test code in the Production-specific projects.

Approach: In order to identify the coupling relationship between changes of different file categories, we analyze for each pair <A, B> of file categories the percentage of commits changing at least one file of category A that also changed at least one file of category B. This percentage corresponds to the confidence of the association rule A=>B. For example, Conf(Infrastructure, InfraTest) measures the percentage of commits changing Infrastructure files that also change InfraTest files. Afterwards, we performed chi-square statistical tests to test whether the obtained confidence values are statistically significant, or are not higher than expected due to chance.

Finally, we also performed qualitative analysis of projects with high coupling to understand the rationale for such coupling. We used "card sorting" (Section 8.3.6) for this analysis. We sampled 50 commits across the top 10 MediaWiki and 50 commits across the top 10 OpenStack projects with the highest Lift metric values (Section 8.3.5) for each coupled pair, i.e., Infra&InfraData, infra&InfraTest, Production&Build, Production&Test. This resulted in 400 commits in total for which we analyzed why IaC and Production files were so tightly coupled with changes to other file categories.

Results: For MediaWiki, Infrastructure files are coupled more tightly with InfraData files, while for OpenStack, Infrastructure files are coupled more tightly with InfraTest files. Figures 8.10a and 8.10c show the coupling between Infrastructure files and InfraData/InfraTest respectively for the Infra-specific group of MediaWiki and OpenStack. The two cases show similar trends to some extent: 31.9% (MediaWiki) and 15.5% (OpenStack) commits, changing InfraData tend to change Infrastructure files as well, while 16.7% (MediaWiki) and 39.2% (OpenStack) commits, changing InfraTest tend to change Infrastructure files. On the other hand, 14.3% commits changing infrastructure files tend to change InfraData file for MediaWiki, while 11.9% commits changing infrastructure files tend to change InfraTest files for OpenStack. This said, Infrastructure files are coupled more tightly with InfraData files than with InfraTest, while for OpenStack we found the opposite: Infrastructure files are coupled more tightly with InfraTest than with InfraData files. This aligns with RQ1, where we found that MediaWiki has a higher proportion of InfraData files and OpenStack has a higher proportion of InfraTest files.

Production files are coupled more tightly with Test files than with Build files. Figures 8.10b and 8.10d show the coupling between Production files and Build/Test files.

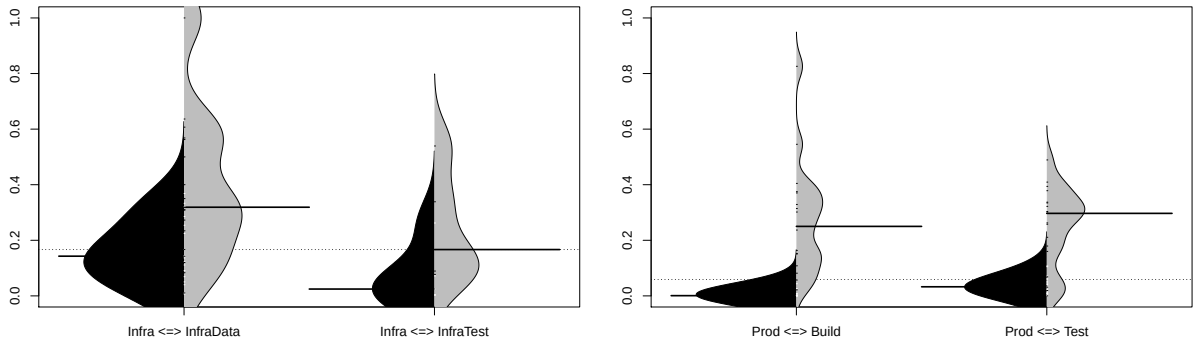
More statistics are shown in Table 8.2. In general, Conf(Production =>Test) has higher value than Conf(Production =>Build), with median of 0.033 for MediaWiki and 0.149 for OpenStack. This indicates Production files are coupled more tightly with Test files than Build files. The low Conf(Production =>Build) might be since OpenStack and MediaWiki are not developed in a compiled language, but instead use Python and PHP, respectively. McIntosh et al. [114] found higher coupling values for Conf(Prod=>Build) in Java-based systems.

Coupling between Infrastructure and InfraData/InfraTest files (for MediaWiki/OpenStack, respectively) in the Infra-specific group is stronger than coupling between Production and Build files in the Production-specific group. Comparing the coupling of Infrastructure-related files and Production-related files in each case study, we can see that the coupling between Infra and InfraData/InfraTest in Infra-specific group (0.143/0.025 for MediaWiki and 0.032/0.119 for OpenStack) is higher than the coupling between Production&Build in the Production-specific group (0.0008 and 0.0012 for MediaWiki and OpenStack respectively). Furthermore, for MediaWiki coupling between Infra&InfraData (median of 0.143) is stronger than Production&Test files (median of 0.033).

Table 8.2 Median confidence values for the coupling relations of code changes (high coupling-more than 0.15 is in bold).

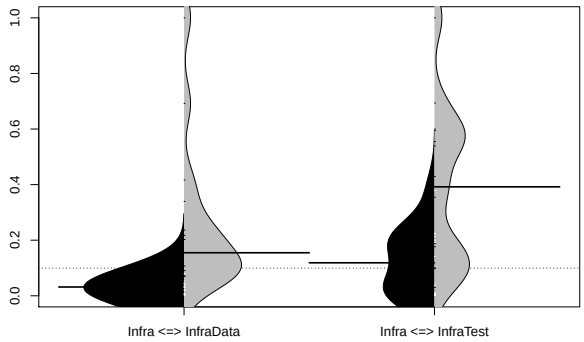
	System	MediaWiki-Infra	OpenStack-Infra
Confidence	Infra =>InfraData	0.143	0.032
	InfraData =>Infra	**0.319**	**0.155**
	Infra =>InfraTest	0.025	0.119
	InfraTest =>Infra	**0.167**	**0.392**
	System	MediaWiki-Production	OpenStack-Production
Confidence	Production =>Build	0.0008	0.0012
	Build =>Production	**0.250**	**0.353**
	Production =>Test	0.033	0.149
	Test =>Production	**0.297**	**0.254**

The most popular reason for coupling of both IaC-related code and Production-related code is "Integration" of new components. Table 8.3 shows the results of our qualitative analysis of the reason why coupling between IaC-related and Production-related code happens. We found a similar trend for both MediaWiki and OpenStack projects. For example, the most popular reason for coupling between Infrastructure and InfraData/InfraTest files in both MediaWiki and OpenStack is "Integration", which indicates adding a new component or service to a project. The second most popular reason is "Modification", i.e., changing a parameter's value or service configuration, which also often causes co-change of

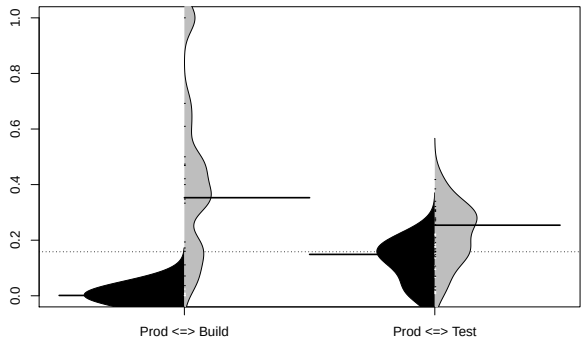


(a) MediaWiki Infra-specific

(b) MediaWiki Production-specific



(c) OpenStack Infra-specific



(d) OpenStack Production-specific

Figure 8.10 Coupling relationship of IaC-related code and Production-related code for MediaWiki and OpenStack.

another file using the same information, representing the second most important reason for coupling.

Despite the similarities between the different kinds of coupling in Table 8.3, we also found differences. In particular, the coupling between production and build code shows a mostly uniform distribution across the four different kinds of coupling. As mentioned earlier, this could be due to the interpreted nature of the programming languages used, with build files mostly used for coordinating packaging, deployment and testing.

Although IaC code seems to be coupled with InfraData and InfraTest for similar reasons as Production and Test code, it is important to realize that regular IDEs have become mature enough to provide tool support for many reasons of coupling of source code. However, to support the same kind of coupling in IaC code, no such tool support currently exists, except for rudimentary syntax highlighting. This is risky, and hence it is necessary to develop such tool support.

Reason	IaC & InfraData		IaC & InfraTest		Prod & Build		Prod & Test	
Initialization	Initialize new environment and add initial files	12%	Initialization of repository	4%	Fork a component/doc import/Initialize repo...	18%	Initialize code and validation check	11%
Integration	Intgrate a new component/database/node, or add new recipes/class/attribute	**46%**	Add new plugin/service/-class/global parameter and basic rspec tests...	**39%**	Add build ability/new target component...	19%	Add new test conditions/validation functions/security suport/new engine/new test framework...	**51%**
Removal	Remove plugin/module/recipe/attribute and clean up installation	15%	Cleanup service manifest/unnecessary recipes...	5%	Move fils to another directory/remove obsolete code...	18%	Remove duplicate functions/unused service and components...	9%
Refactoring	Refactor component (e.g., move patches to upstream) or reorganize cookbook	6%	Refactor server configuration/perform DB migration...	8%	Rearrange package structure/migrate build config...	3%	Change code structure (e.g., separate API and implementation)...	3%
Upgrade	Update dependencies or environment set-up to fit new requirement	6%	Update temporary fixes or configuration (e.g., permission mode)	9%	Update API version/front layout/fully merge/roll back...	6%	N/A	0
Modification	Change environment configuration (e.g., authorization/certificate)	12%	Change role description/service configuration, improve error handling, more test options, fix broken test.	**22%**	Change new API/dir...	15%	Fix test cases or change service functionalities/update database version...	10%
Textual Edit	Add author/license... information.	3%	Rename/normalize files, add comment, move attribute to another space...	13%	Update license/copyright/format or rename files...	**21%**	Fix typos/remove copyright/ fix license headers/rename plugin's config...	16%

Table 8.3 The reasons for high coupling, based on a sample of 400 (50*4*2) commits for MediaWiki and OpenStack.

For MediaWiki, Infrastructure files are coupled more tightly with InfraData files, while for OpenStack, Infrastructure files are coupled more with InfraTest files. Coupling for Infrastructure and InfraData/InfraTest files is higher than that of Production and Build files for both MediaWiki and OpenStack, while coupling for Infrastructure and InfraTest files is higher than that of Production and Test files for MediaWiki as well. Furthermore, the most popular reason for such coupling is "Integration" of a new component or service.

RQ5: Who changes infrastructure code?

Motivation: Herzig et al. [90] studied the impact of test ownership and team social structure on the testing effectiveness and reliability and found that they had a strong correlation. This means that in addition to the code-change relations of RQ4, we also need to check the relationship between the developers, i.e., the ownership among different file categories. Even though test or code changes might require IaC changes, it likely makes a difference whether a regular tester or developer makes such a change compared to an IaC expert.

Approach: First, we need to identify the ownership for each category. For this, we check the author of each commit. If a commit changes an infrastructure file, then we identify that commit's author as infrastructure developer. An author can have multiple identifications, e.g., one can be an infrastructure file and production developer at the same time (even for the same commit). We ignore those developers whose commits only change the files of the "Other" category.

We then compute the RQ4 metrics, but this time for the change ownership. Conf(Infrastructure, Build) is the percentage of IaC developers that also change build files out of the total number of developers changing at least once infrastructure file.

Results: More Infrastructure developers change InfraData files than change InfraTest code. Figures 8.11a and 8.11c show how for MediaWiki and OpenStack, coupling for Infrastructure and InfraData is higher (median of 0.17 and 0.10 for MediaWiki and OpenStack respectively) than coupling for Infrastructure and InfraTest (median of 0.08 and 0.07 respectively), while most of developers changing InfraData code also change Infrastructure

code (median of 0.76 and 1.00 respectively), while for OpenStack, half of developers changing InfraTest code change Infrastructure code as well.

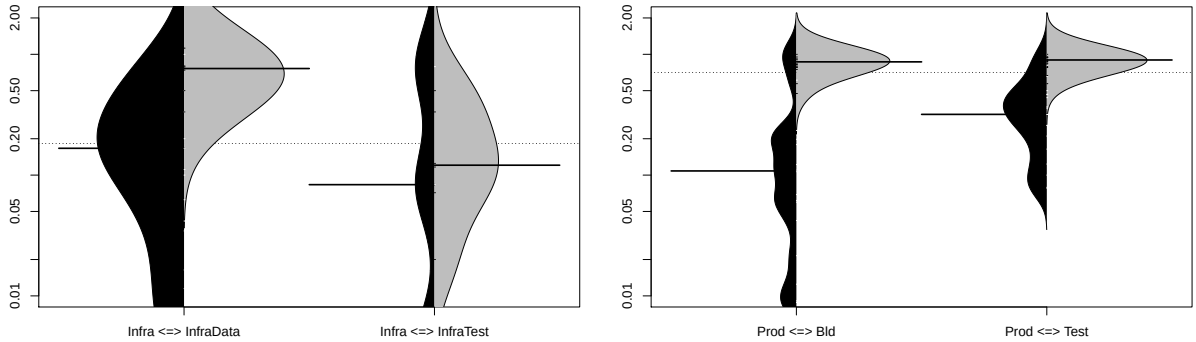
Production developers change Test code as well, and 11% Production developers change Build code as well for MediaWiki. Figure 8.11 and Table 8.4 show for the Production-specific group that 32% (MediaWiki) and 74% (OpenStack) Production developers also change Test code. 11% Production developers change Build code for MediaWiki, while for OpenStack, Production and Build are not coupled that much in terms of ownership (median of 0.05).

Coupling between Infrastructure and InfraData developer ownership is stronger to that of Production and Build code, but less than that of Production and Test code. We observed that coupling between Infrastructure and InfraData in terms of ownership for both MediaWiki (median of 0.17) and OpenStack (median of 0.10) is higher than coupling between Production and Build (median of 0.11 and 0.05 for MediaWiki and OpenStack respectively), while lower than coupling between Production and Test code (median of 0.32 and 0.74 respectively). On the other hand, coupling between Build/Test and Production (median of 0.87/0.90 and 1.00/0.95 respectively) is higher than coupling between InfraData/InfraTest and Infrastructure (median of 0.76/0.12 and 1.00/0.53).

Table 8.4 Median confidence values for the coupling relations involving IaC developers.

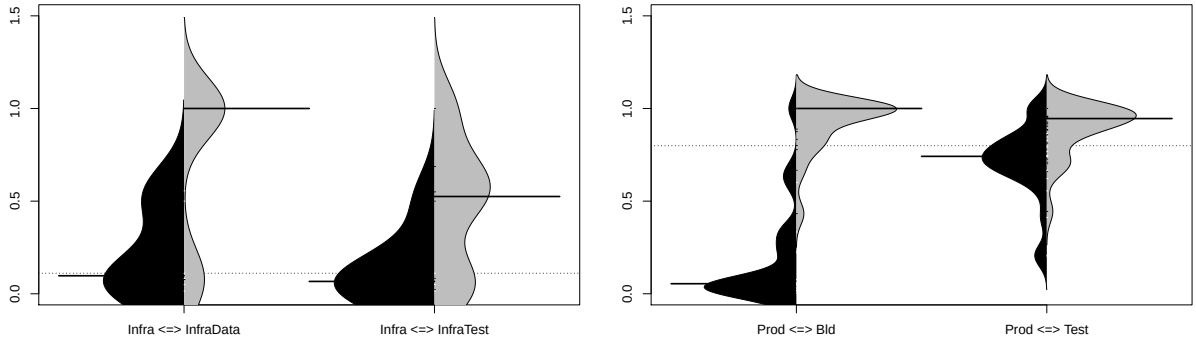
	System	MediaWiki-Infra	OpenStack-Infra
Confidence	Infra =>InfraData	0.17	0.10
	InfraData =>Infra	**0.76**	**1.00**
	Infra =>InfraTest	0.08	0.07
	InfraTest =>Infra	0.12	**0.53**
	System	MediaWiki-Production	OpenStack-Production
Confidence	Production =>Build	0.11	0.05
	Build =>Production	**0.87**	**1.00**
	Production =>Test	0.32	**0.74**
	Test =>Production	**0.90**	**0.95**

InfraData developers also change InfraData files. Ownership coupling between Infrastructure and InfraData in the Infra-specific group is stronger than that of Production and Build developers' ownership.



(a) MediaWiki Infra-specific

(b) Mediawiki Production-specific



(c) OpenStack Infra-specific

(d) OpenStack Production-specific

Figure 8.11 Coupling of ownership infra-specific (log-scaled).

8.5 Threats To Validity

Construct validity threats concern the relation between theory and observation. First of all, we use the confidence of association rules to measure how closely file categories and owners co-change in our research questions, similar to earlier work [114]. Furthermore, we use the monthly churn per file to measure the frequency of change and the number of changed lines of code to measure the amount of change. However, these metrics might not 100% reflect the actual coupling relationship and churn rate. Other metrics should be used to replicate our study and compare findings.

Another threat to construct validity concerns the impact of the manual classification on our results. For this classification, we adopted the semi-automatic approach of McIntosh et al. [114], consisting of a script to separate certain file categories, then manually classifying the remaining files. To mitigate bias, at first the first author of this paper did this classification, followed by an independent verification by the second author.

Threats to external validity concern the ability to generalize our results. Even though two

large open source ecosystems have been studied, we cannot generalize our findings to other open and closed source projects. On the other hand, because OpenStack and MediaWiki consist of a variety of projects, and also have adopted the two most popular infrastructure tools (Puppet and Chef), they are representative case studies to analyze. However, both of them use interpreted programming languages, and they do not use the same conventions regarding whether or not production and infrastructure code can be mixed in the same repository. Further studies should consider other open and closed source systems.

Another threats to external validity concerns the use cases of IaC in different systems. Different projects may use IaC ion different purpose, such as deploying applications, or integration testing. This may bias our analysis, which ignores the role played by IaC in the analyzed systems.

8.6 Conclusion

IaC (Infrastructure as Code) helps automate the process of configuring and populating the environment in which the software product will be developed, tested and/or deployed. The developed files that specify how to create the environment are a kind of source code files in a dedicated programming language, managed under version control. Ideally, this practice helps simplify the automatic generation of environments, shorten the release cycle, and reduce the possible inconsistencies introduced by manual work, however the amount of maintenance required for this new kind of source code file is unclear.

We empirically studied this maintenance effort and compared the evolution process for Infrastructure-related code to that of Production-related code in the context of the OpenStack and MediaWiki ecosystems. These are large-scale open source ecosystems that have adopted infrastructure-as-code since a couple of years. We studied 50 projects for OpenStack and 48 projects for MediaWiki.

We found that the analyzed infrastructure-heavy projects have a median value of 9 and 60 IaC files for MediaWiki and OpenStack respectively, which yields a large proportion in each (median value of 55% and 52% respectively). These IaC files are larger than infrastructure-related data and test files, with a median value of 47 and 76 lines of code for MediaWiki and OpenStack. 11.8% and 25% of the infrastructure files of MediaWiki and OpenStack projects change monthly, which is higher than all other file categories for OpenStack and higher than infrastructure-related data and test files for MediaWiki. The monthly churn size of infrastructure files is not significantly higher than production files, but after being normalized by the number of IaC and production files, infrastructure files have a higher

relative churn than production files (with median value of 2.03 and 0.240 lines of code for MediaWiki and OpenStack respectively, compared to 0.02 and 0.024 of production files). The large relative churn size of Infrastructure files suggests potential high maintenance effort.

Furthermore, for MediaWiki Infrastructure files are coupled tightly with InfraData files in Infra-specific group while for OpenStack, Infrastructure files are coupled more tightly with InfraTest files. Both tend to be more tightly than coupling between Production and Build files. Additionally, the most popular reason for such coupling is "Integration" of new components or services. Finally, coupling between Infrastructure and InfraData developers is stronger than that between Production and Build developers.

Taking all these findings together, we believe that IaC files should be considered as source code files, not just because of the use of a programming language, but also because their characteristics and maintenance show similar symptoms as source code files. Hence, more work is necessary to study bug-proneness of IaC files as well as help reduce the maintenance effort.

Furthermore, whereas IDE and other tool support exists to help developers deal with coupling, no such support exists for IaC, hence future work should also focus on this.

8.7 Appendix

Table 8.5 The proportion of the six file categories for Infra- and Production-specific group of MediaWiki and OpenStack.

			Production	Build	Test	Infrastructure	InfraTest	InfraData
MediaWiki	Infra-specific	1st Qu.	0.00	0.00	0.00	0.32	0.00	0.025
		Median	0.01	0.00	0.00	0.55	0.00	0.233
		Mean	0.15	0.017	0.05	0.49	0.03	0.267
		3rd Qu.	0.12	0.06	0.00	0.72	0.00	0.411
	Production-specific	1st Qu.	0.52	0.005	0.08	0.00	0.00	0.00
		Median	0.76	0.007	0.22	0.00	0.00	0.00
		Mean	0.64	0.043	0.29	0.005	0.001	0.02
		3rd Qu.	0.83	0.009	0.42	0.00	0.00	0.00
OpenStack	Infra-specific	1st Qu.	0.01	0.01	0.00	0.05	0.00	0.00
		Median	0.16	0.02	0.03	0.52	0.01	0.01
		Mean	0.28	0.03	0.07	0.45	0.12	0.04
		3rd Qu.	0.70	0.03	0.15	0.68	0.24	0.02
	Production-specific	1st Qu.	0.66	0.001	0.16	0.002	0	0.00
		Median	0.72	0.003	0.28	0.003	0	0.00
		Mean	0.72	0.008	0.26	0.005	0	0.0005
		3rd Qu.	0.79	0.008	0.34	0.005	0	0.00

Table 8.6 Distribution of file size (LOC) for the MediaWiki and OpenStack projects respectively.

			Production	Build	Test	Infrastructure	InfraTest	InfraData
MediaWiki	Infra-specific	1st Qu.	0	0	0	35	0	7
		Median	37	0	0	47	0	20
		Mean	125	20	26	80	12	31
		3rd Qu.	236	30	0	120	0	44
	Production-specific	1st Qu.	137	39	53	0	0	0
		Median	216	78	91	0	0	0
		Mean	205	132	101	26	89	7
		3rd Qu.	283	164	153	0	0	0
OpenStack	Infra-specific	1st Qu.	8	30	0	40	0	0
		Median	29	68	39	76	41	25
		Mean	79	90	61	81	48	32
		3rd Qu.	99	93	89	138	72	58
	Production-specific	1st Qu.	82	75	141	44	0	0
		Median	117	104	215	78	0	0
		Mean	134	102	196	95	0	4
		3rd Qu.	166	146	265	138	0	0

Table 8.7 The proportion of changed files per month the MediaWiki and OpenStack projects respectively.

			Production	Build	Test	Infrastructure	InfraTest	InfraData
MediaWiki	Infra-specific	1st Qu.	0.239	0.061	0.129	0.116	0.023	0.016
		Median	0.247	0.073	0.132	0.118	0.036	0.019
		Mean	0.234	0.096	0.134	0.106	0.034	0.016
		3rd Qu.	0.254	0.081	0.133	0.122	0.047	0.020
	Production-specific	1st Qu.	0.245	0.128	0.119	0.043	0.002	0.009
		Median	0.277	0.132	0.123	0.050	0.013	0.016
		Mean	0.273	0.144	0.122	0.059	0.020	0.014
		3rd Qu.	0.307	0.141	0.129	0.074	0.043	0.018
OpenStack	Infra-specific	1st Qu.	0.190	0.222	0.149	0.243	0.096	0.120
		Median	0.201	0.228	0.163	0.250	0.115	0.134
		Mean	0.202	0.239	0.186	0.252	0.128	0.137
		3rd Qu.	0.217	0.249	0.226	0.266	0.136	0.148
	Production-specific	1st Qu.	0.214	0.162	0.196	0.197	0.052	0.064
		Median	0.229	0.174	0.232	0.199	0.058	0.073
		Mean	0.231	0.180	0.225	0.209	0.061	0.078
		3rd Qu.	0.245	0.193	0.251	0.224	0.70	0.090

Table 8.8 The distribution of monthly churn per project for the MediaWiki and OpenStack projects respectively

			Production	Build	Test	Infrastructure	InfraTest	InfraData
MediaWiki	Infra-specific	1st Qu.	1.00	8.78	1.00	17.56	0.00	0.0
		Median	1.00	14.16	1.00	20.01	0.00	1.0
		Mean	8.19	20.51	49.07	23.58	8.80	7.46
		3rd Qu.	9.50	26.74	61.87	28.37	1.00	2.13
	Production-specific	1st Qu.	14.21	0.00	27.54	0.00	16.77	0.00
		Median	19.56	1.40	33.04	0.00	29.77	0.00
		Mean	44.06	2.17	49.52	2.57	27.87	12.88
		3rd Qu.	39.43	1.89	39.75	1.00	36.33	1.00
OpenStack	Infra-specific	1st Qu.	10.52	1.65	15.82	16.34	2.00	0.00
		Median	20.30	24.50	34.22	21.97	25.45	19.49
		Mean	27.97	17.66	78.82	29.06	30.21	17.92
		3rd Qu.	48.00	24.94	58.53	46.01	59.54	27.77
	Production-specific	1st Qu.	19.08	0.00	30.01	17.28	28.84	0
		Median	31.96	1.00	36.81	28.17	44.38	0
		Mean	36.56	3.317	56.67	28.36	57.77	0.67
		3rd Qu.	45.00	1.00	69.53	37.00	63.04	1

Table 8.9 The distribution of MCF value for the MediaWiki and OpenStack projects respectively.

			Production	Build	Test	Infrastructure	InfraTest	InfraData
MediaWiki	Infra-specific	1st Qu.	0.46	2.00	0.00	0.27	0.00	0.11
		Median	1.00	9.24	3.70	2.03	1.00	0.16
		Mean	2.02	12.36	39.91	2.44	3.85	1.26
		3rd Qu.	1.00	16.34	25.64	2.77	2.10	0.63
	Production-specific	1st Qu.	0.01	0.07	0.058	1.0	12.10	0.00
		Median	0.02	0.11	0.118	1.0	18.83	2.20
		Mean	1.16	0.20	0.360	1.46	22.37	12.80
		3rd Qu.	0.05	0.28	0.265	2.0	36.02	5.06
OpenStack	Infra-specific	1st Qu.	0.059	0.33	0.251	0.085	0204	0.083
		Median	1.00	0.61	1.00	0.240	1.00	0.241
		Mean	3.17	2.34	53.71	0.446	20.24	7.672
		3rd Qu.	4.06	4.16	10.50	0.583	52.68	19.493
	Production-specific	1st Qu.	0.011	0.11	0.090	2.50	28.84	1.00
		Median	0.024	0.17	0.116	3.94	44.38	1.00
		Mean	0.044	0.51	0.757	5.32	57.77	0.93
		3rd Qu.	0.073	1.00	0.278	6.40	63.04	1.00

CHAPTER 9 GENERAL DISCUSSION

In this thesis, we conducted a series of empirical studies on two critical phases of Release engineering: Integration and Provisioning (Infrastructure-as-Code), in order to help practitioners understand the progress of Release engineering and evaluate state-of-the-art tools. Integration is one of the most complex phases of software engineering, while Infrastructure-as-Code is one of the most recent techniques. Furthermore, ample repository data is available about these two phases.

This chapter discusses the main contributions and findings of this thesis.

9.1 Explanatory Models in Integration Process

We studied the Integration process of the Linux kernel project in Chapter 4. While many prior studies have been done about reviewing and integration process individually, our work was the first to link data of reviewing and integration process together and analyze the overall progress. In this chapter, we build the explanatory model to understand the reviewing and integration process. Our model for simulating patch acceptance has high performance with up to 74% precision and recall, and models for integration and reviewing time have an accuracy of up to 70% and 76% respectively.

While explanatory models are not able to predict future acceptance or acceptance time, only to explain the existing data, the good performance of such models paves the way for building prediction models with reasonable performance. Of course, such prediction models could introduce the risk of people "gaming" the system by abusing the knowledge in these models. This is why we instead preferred explanatory models, since those provides key insights of "good" patches that could be turned into best practices for the field.

Thus far, the work in Chapter 4 has had 37 citations, where people have been extending their work in the areas of the impact of reviewing process on software quality, management of community contributions and development decision-making analysis.

9.2 Major Factors Influencing the Integration Process

In Chapter 4, we did quantitative analysis of the reviewing and integration process and proposes three lists of factors that impact Integration acceptance ratio, Reviewing time, and Integration time. For example, five key factors related to acceptance ratio are msg_exp

(number of patches sent by authors before), commit_exp (number of commits accepted thus far by the author), thr_part (number of people participating in current thread until current patch), nth_try (revision number of this patch), and commit_sub (prior churn). While commit_sub is related to the history of changed subsystem, and hence cannot be controlled by developers, msg_exp and commit_exp are relative to developer experience, and thr_part and nth_try are relative to patch maturity. Hence, developers can learn that gaining relative experience and developing more mature commits will increase the probability of getting accepted.

Besides, we also provide many suggestions for developers to help increase the probability of getting their patches accepted by enhancing the interaction with the OSS community, or splitting a larger patch into smaller pieces. For maintainers, they can give high priority to those patches that are submitted by more experienced developers, or those commits that have gone through multiple discussions.

9.3 Novel Linking Approaches for Low-tech Reviewing System

In Chapter 5, we propose three different approaches that link (email, email) pairs and (email, commit) pairs in the same thread. The three approaches include the token-based approach (least strict, most fine-grained), plus-minus-line-based technique (medium strict, medium fine-grained), and checksum-based approach (most strict, least fine-grained). We applied these three techniques in the dataset of the Linux kernel and found that the plus-minus-line-based approach achieves the best performance in linking (email, email) pairs while the combination of plus-minus-line-based approach and checksum-based approach achieves the best performance in linking (email, commit) pairs.

In our prior empirical study about the Integration process on the Linux kernel (Chapter 4), we used the checksum-based approach to recover reviewing threads. According to our findings in Chapter 4, the use of this technique instead of the plus-minus-line technique might have missed links between some emails and commits. This might make our findings about Integration process different. However, as the checksum-based approach is the most strict, although we may miss some correct pairs, most of the linked pairs are correct. This could be seen when we evaluated the precision of the checksum-based approach with 100 random selected emails across all mailing lists, and found that had a precision of 100%±10%. Hence, as models of integration process might not be using complete data, they do use accurate data.

In addition, among all impact factors that influence the Integration process, only “nth_try”

(revisions a patch has) might be affected by not using the plus-minus approach to link patch revisions in different threads to each other. For that factor, we used the naming convention used in the subject of a thread to link threads together, which is acceptable (and was used as oracle in Chapter 5), but not perfect. Given that the heuristic we used in Chapter 5 misses to link certain threads, our values for nth_try are lower than the real value. However, since our findings show that a higher value of "nth_try" leads to higher acceptance probability and shorter integration time, if we change to the plus-minus-line-based approach, the value of "nth_try" will only be higher, and hence will not change our conclusion. Therefore, while we consider to apply a more valid link approach to re-analyze the Integration process in Chapter 4 in future work, the current findings still hold.

9.4 Empirical comparison of Different Integration Approaches

Chapter 6 presents our work about the impact of the toggling technique on integration process. We conducted an empirical study on a real commercial project in a company. This project started with a traditional branching structure, but gradually added toggles in their source code after one year of development. This enables us to observe the difference before and after adopting the toggling technique in one and the same project, avoiding noise caused by changing case study context. Given that toggling is only fairly recently being used as replacement for branching, and this is mostly done by industry, our analysis is based on a unique data set. In addition, the good performance on the hybrid integration structure shows other companies a promising solution of migrating to a new technique. However, we also consider to conduct empirical studies on the pure toggling-based integration process to explore more deeply this integration technique and how it impacts on Release engineering. For now, access to a relevant (industrial) dataset is the main challenge.

Our case study and Google Chrome in [131] both use toggles to guard features developed by different teams. While in our case study toggles are used as an alternative of branching used to do integration, paper [131] empirically studied toggling usage in Google Chrome where toggles are used for enabling features for testing and release. Toggling as an alternative for branching might not be applicable in other large open source software systems, such as the Linux kernel. The Linux kernel project accepts contribution from developers all over the world and many of them are individuals instead of a team. Thus, the submitted patch usually changes small code block such as a function or even a typo instead of contributing a large and complete feature. Using toggles to guard such individual patches risks requiring too many toggles for these patches, which introduce major code complexity and technical debt. Hence, even though the Linux kernel uses toggles to select which features to compile into the

kernel, using toggles to replace traditional branch-based integration in such large-scale OSS projects needs further research.

9.5 Understand Characteristics of Infrastructure-as-Code (IaC)

It turned out that IaC code takes large proportion in OpenStack, and changes frequently, with large churn size. IaC code changes practically co-evolve with production code change. In MediaWiki, IaC code shows similar phenomena in terms of code changes. The most popular reason for IaC code co-changes is "Integration of new modules/service". All of this indicates that the effort needed to maintain IaC code is underestimated. IaC code should be considered separately and deserves more attention.

We also found IaC code is lightly coupled with IaC test and data files. This coupling is more hidden and hence more risky, as it is easily ignored.

9.6 Datasets of IaC File Classification Lists

In Chapter 8, we studied the characteristics of IaC code and the coupling relationship between IaC code change and source code change of OpenStack. We classified all files into five categories: IaC code, Production code, Build, Test and Other. Furthermore, in Chapter 7 we studied the evolution of IaC-related code and compared with the evolution of Production-related code of OpenStack and MediaWiki. In this paper, we classified all file revisions into more detailed categories: IaC-related files (IaC code, IaC test and IaC data files), and Production-related files (Production code, Build and Test files). All these classifications are available online to encourage more researchers conduct empirical studies on IaC techniques.

Finally, our work about Infrastructure-as-Code (IaC) was the first empirical study ever on this topic. Inspired by our work, Sharma et al. [144] performed analysis about quality of infrastructure code.

Release engineering has developed enormously and been becoming mainstream in industry. However, comparatively little empirical research of mining software repositories has been done in this field. Our empirical studies have covered two critical phases of Release engineering, where repository data is available. By our empirical research we desire to understand the progress of Release engineering, and help practitioners gain sense of effort and expense they need to pay for adopting the modern techniques.

Although, as is, our findings cannot be generalized to the whole release engineering and all phases, we believe our studies present insight in Release engineering and propose considerable

potential for future research.

CHAPTER 10 CONCLUSION AND FUTURE WORK

Modern Release engineering (Releng) aims to provide an automated and smooth pipeline to deliver high quality software products to end users as soon as possible. Release engineering consists of several phases, from "Integration", "Building", "Testing', "Provisioning (Infrastructure-as-Code)" until finally "Deployment" and "Release". Each phase should be fully automated and, apart from some testing phases (such as UI testing), to get rid of human labor involved, to increase release velocity and reduce the noise introduced by manual work.

A myriad of new techniques/tools keep on emerging to meet this demand of high speed and low cost in Release engineering. Choosing one of the most suitable technique becomes a problem for many communities especially for smaller companies or start-ups, where migrating between techniques is very costly. To take full advantage of Release engineering requires understanding the progress of Release engineering and how these new techniques work in practice. However, research about related topics is still lacking. Especially for new Release engineering techniques such as Infrastructure-as-Code, our work is the first empirical study about IaC. Hence, in this thesis we argued for using mining software repositories to analyze the progress of Release engineering phases, as well as help practitioners evaluate state-of-the-art Releng techniques. We focused on the Integration and Provisioning (Infrastructure-as-Code) phases of Release engineering, where ample repository data is available.

10.1 Understanding the progress of Integration

Integration is one of the most complex phases in software development. A complete product is basically an assembly of contributions from all developers, integrated in a coordinated way. Hence, an efficient integration approach will help significantly decrease the cost of Release engineering in terms of time (delays) and hence money. However, current Integration process bears many problems, such as integration conflicts and lack of transparency for contributors. Integration conflicts are caused by developers working in parallel changing the same code block, and take developers effort to figure out a solution and to fix them.

In addition, in large open source software (OSS) communities, the Integration process is not completely transparent and visible to contributors, but under control of a small group of core maintainers. Contributors involved with the Review process are unaware of what happens after his patch is integrated by a subsystem maintainer until finally included in the new release (if all goes well). Furthermore, in many large OSS communities (e.g., the Linux and

Apache), the review process is still using a low-tech environment, where patches are submitted by sending emails to maintainers. In this low-tech review environment, it is hard to track the complete reviewing history of a patch, especially when it underwent multiple versions, for lacking a physical link between patches (emails) and commits. New techniques such as Toggling aim to increase the speed of integration to avoid integration conflicts. However, no evidence yet has shown quantitatively that this technique works as expected.

We selected the Linux kernel as the case study to study the performance of reviewing and integration processes (Chapter 4). We linked reviewing and integration data together and analyzed what factors may have an impact on them. We found that smaller, more mature patches targeting the right subsystems may get accepted more possibly. Meanwhile, reducing the number of subsystems affected by the patch and being more active in the community can help shorten reviewing and integration time.

During our study, we found out that a major challenge for MSR analysis is linking data from different repositories. We proposed a checksum-based technique to link patches from mailing lists and commits from git repositories together to recover the whole reviewing and integration process. This approach has a high precision. However, we then proposed three approaches with different granularity and strictness (Chapter 5). We found that the plus-minus-line-based technique found the most links between patch revisions in different threads, while the combination of the line-based technique and the checksum-based technique achieves the best performance in linking reviewing and integration process.

These recovered links allowed us to find that 25% of patches have a reviewing history longer than four weeks, and patches evolved multiple versions are easier to get accepted. These novel link approaches will help (1) project communities migrate to another modern reviewing system, and (2) researchers to study code changes with a complete history.

10.2 Evaluating an Integration Tool

Chapter 6 introduced a technique that aims to make integration more continuous to get rid of problems in previous integration process based on traditional branching technique. Toggles enable teams to work on the same branch, turning off toggles to make sure on different features do not disturb each other. Yet, no quantitative evidence shows that Toggling can really improve integration process efficiency. Hence, we empirically study a commercial project that adopts this new technique. As it is risky to get rid of all branches and migrate to a pure toggle-based integration, this project adopted a hybrid structure that combines toggles with branching integration structure. By mining repositories, we found that this hybrid

integration approach helps decrease integration effort and improve productivity significantly. Hence, this practice is a good idea for those companies wanting to try out this new technique.

10.3 Understanding the Evolution of IaC

IaC is one of the latest techniques for automating the process of provisioning a new environment (infrastructure, e.g. a virtual machine), where an application can be compiled, tested, and deployed. However, as IaC can be considered as source code that can be compiled, tested, and version controlled, adopting this new technique requires maintenance effort. Hence, it is vital to understand the potential effort required before deciding to use this new technique.

In Chapter 7 we conducted an empirical study on open source software project OpenStack to study the characteristics and maintenance effort of IaC code. We collected all file revisions from OpenStack git repositories and classified them into four categories: IaC code, Production, Build and Test files. We found that IaC code files tend to take up a large proportion and churn frequently in large size. In addition, the code change is coupled tightly with that of source code. This implies substantial maintenance effort. The most popular reason for this coupling is "Integration of new modules/frameworks".

To understand more deeply the evolution process of IaC code, we conducted another empirical study on two case studies: OpenStack and MediaWiki (Chapter 8). This time, we classified all files into more detailed categories, basically two dimensions: IaC-related (IaC code, IaC test and IaC data) files, and Production-related (Source code, Build and Test) files. We tried to understand if IaC related code evolves similarly as Production-related code. Besides general patterns such as IaC code still takes up a large proportion and change frequently, we observed that IaC code files are lightly coupled with IaC test and data files. This hidden relationship may be dangerous as developers hardly notice the hidden relationship. Based on our findings, IaC code should be considered as normal source code that should deserve more attention and support.

Overall, we found that by mining software repositories we gain more sense about the *Integration* and *Provisioning* two phases, as well as measuring how state-of-art techniques work in practice. Meanwhile, additional MSR analysis should be performed to discover the proper techniques and tools to improve the whole Release engineering process, and provide more constructive advice to practitioners.

10.4 Future Work

Our research is not perfect and we will improve in our future work the following aspects:

10.4.1 Re-analyzing Linux Integration Process with Other Linking Techniques

In our first empirical study about Integration process, we used the checksum-based technique, which is strict and in a coarse-grained. However, in our following work, which compared three different linking techniques, we found that a line-based approach can find more links. This does not affect the correctness of our conclusion but causes missing information, hence we would like to apply this technique to redo the empirical analysis to find out more characteristics.

10.4.2 Exploring Toggling Technique More in Practice

In our case study about Toggling, toggles are used as guard of features in the integration process. However, toggles can be used in various way, e.g., to help testing and release. We would like to study toggling technique in different usage in other Release engineering phases.

Furthermore, our case study project adopted a hybrid structure combining toggling and branching techniques. We want to analyze how Toggling works in a pure setting to analyze if it still improves the efficiency of Release engineering.

10.4.3 Understanding the Coupling Relationship Between Source Code and Resource Data Files.

In our empirical study about IaC code, we studied the coupling relationship between IaC code and IaC configuration data files. This inspires us to explore the relationship between source code and general resource data files (e.g., pictures, textual configuration, videos). If source code are coupled with other code files, we would notice this coupling by reading source code. However, we hardly know which data resource files have been changed when we change source code. How much effort is evolved by this co-change might be a good research topic in future.

10.4.4 More Case Studies on Integration and Infrastructure-as-Code

In this thesis, we conducted empirical studies on *Integration* and *Infrastructure-as-Code* with case studies of the Linux kernel, OpenStack and MediaWiki respectively. In our future work,

we desire to apply the same approach on other case studies to validate and generalize our findings.

10.4.5 Empirical Study on Other Release Engineering Phases

Besides Integration and Infrastructure-as-Code, Release engineering also includes other phases like Continuous Integration, Building, Deployment and Release. Each phase has an important impact on the whole Release engineering process. Hence, an empirical study of each phase will help improve Release engineering from different perspectives.

BIBLIOGRAPHY

[1] Ansible. https://www.ansible.com/.

[2] Ant. http://ant.apache.org/.

[3] Buildbot. http://buildbot.net/.

[4] Cfengine. https://cfengine.com/.

[5] chefspec. https://docs.chef.io/chefspec.html.

[6] Docker. https://www.docker.com/.

[7] Feature toggle bliki by martin fowler. http://martinfowler.com/bliki/FeatureToggle.html.

[8] Feature toggles are one of the worst kinds of technical debt. https://dzone.com/articles/feature-toggles-are-one-worst.

[9] Git. https://git-scm.com/.

[10] Infrastructureascode. http://martinfowler.com/bliki/InfrastructureAsCode.html.

[11] Jenkins. https://jenkins.io/.

[12] Kruskal-wallis test. http://www.r-tutor.com/elementary-statistics/non-parametric-methods/kruskal-wallis-test.

[13] Maven. https://maven.apache.org/.

[14] mercurial-scm. https://www.mercurial-scm.org/.

[15] Msr 2015. http://2015.msrconf.org/.

[16] Openstack git repository browser. https://git.openstack.org.

[17] Periodic table of devops tools. https://xebialabs.com/periodic-table-of-devops-tools/.

[18] puppetlabspuppetlabs-openstack. https://github.com/puppetlabs/puppetlabs-openstack.

[19] Recommending useful software artifacts. https://www.cs.ubc.ca/labs/spl/projects/hipikat/.

[20] Relative origin of cfengine, puppet and chef. http://verticalsysadmin.com/blog/relative-origins-of-cfengine-chef-and-puppet/.

[21] Release engineering international workshop. http://releng.polymtl.ca/RELENG2016/html/index.html.

[22] Repositories of wikimedia foundation. https://github.com/wikimedia.

[23] Repositories of wikimedia foundation. https://www.youtube.com/watch?v=j-9-zuV5JNo.

[24] rspec-puppet. http://rspec-puppet.com/.

[25] stackforgeopenstack-chef-repo. https://github.com/stackforge/openstack-chef-repo.

[26] Strider. https://github.com/Strider-CD/strider.

[27] Subversion. https://subversion.apache.org/.

[28] Travisci. https://travis-ci.org/.

[29] *An American National Standard, IEEE Standard for Software Unit Testing.* IEEE computer society. Software engineering technical committee, 1986.

[30] B. Adams, S. Bellomo, C. Bird, T. Marshall-Keim, F. Khomh, and K. Moir. The practice and future of release engineering: A roundtable with three release engineers. *IEEE Software*, 32(2):42–49, Mar 2015.

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

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

[33] B. Al-Ani, E. Trainer, R. Ripley, A. Sarma, A. van der Hoek, and D. Redmiles. Continuous coordination within the context of cooperative and human aspects of software engineering. In *Proceedings of the 2008 International Workshop on Cooperative and*

Human Aspects of Software Engineering, CHASE '08, pages 1–4, New York, NY, USA, 2008. ACM.

[34] J. M. Al-Kofahi, H. V. Nguyen, A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Detecting semantic changes in makefile build code. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 150–159, Sept 2012.

[35] J. Andrews. Linux: Cml2, esr & the lkml. `http://kerneltrap.org/node/17`, February 2002.

[36] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement?: A text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, pages 23:304–23:318, New York, NY, USA, 2008. ACM.

[37] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 361–370, New York, NY, USA, 2006. ACM.

[38] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 298–308, Washington, DC, USA, 2009. IEEE Computer Society.

[39] L. Aversano, L. Cerulo, and C. Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, IWPSE '07, pages 19–26, New York, NY, USA, 2007. ACM.

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

[41] A. Bacchelli, T. Dal Sasso, M. D'Ambros, and M. Lanza. Content classification of development emails. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 375–385, Piscataway, NJ, USA, 2012. IEEE Press.

[42] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: Bugs and bug-fix commits. In *Proceedings of the Eighteenth ACM SIGSOFT*

International Symposium on Foundations of Software Engineering, FSE '10, pages 97–106, New York, NY, USA, 2010. ACM.

[43] L. Barker. Android and the linux kernel community. `http://www.steptwo.com.au/papers/kmc_whatisinfoarch/`, May 2005.

[44] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu. Cohesive and isolated development with branches. In *Proc. of the 15th intl. conf. on Fundamental Approaches to Software Engineering (FASE)*, pages 316–331, 2012.

[45] K. M. Bartol. Motivating and managing computer personnel: by j. d. cougar and r. a. zawacki (new york; wiley 1980). *Information & Management*, 4(5):281–282, 1981.

[46] L. Bass, I. Weber, and L. Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 1st edition, 2015.

[47] O. Baysal, I. Davis, and M. W. Godfrey. A tale of two browsers. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 238–241, New York, NY, USA, 2011. ACM.

[48] O. Baysal, R. Holmes, and M. W. Godfrey. Mining usage data and development artifacts. In *Proc. of the 9th IEEE working conf. on Mining Software Repositories (MSR)*, pages 98–107, 2012.

[49] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. The influence of non-technical factors on code review. In R. Lämmel, R. Oliveto, and R. Robbes, editors, *WCRE*, pages 122–131. IEEE, 2013.

[50] S. Beecham, N. Baddoo, T. Hall, H. Robinson, and H. Sharp. Motivation in software engineering: A systematic literature review. *Inf. Softw. Technol.*, 50(9-10):860–878, Aug. 2008.

[51] N. Bettenburg, A. E. Hassan, B. Adams, and D. M. German. Management of community contributions a case study on the android and linux software ecosystems. *Empirical Software Engineering*, 20(1):252–289, Feb. 2015.

[52] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 27–30, New York, NY, USA, 2008. ACM.

[53] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan. An empirical study on inconsistent changes to code clones at the release level. *Sci. Comput. Program.*, 77(6):760–776, 2012.

[54] N. Bettenburg, E. Shihab, and A. E. Hassan. An empirical study on the risks of using off-the-shelf techniques for processing mailing list data. In *Proc. of the 25th IEEE Intl. Conf. on Software Maintenance (ICSM)*, pages 539–542, 2009.

[55] N. Bettenburg, S. W. Thomas, and A. E. Hassan. Using fuzzy code search to link code fragments in discussions to source code. In T. Mens, A. Cleve, and R. Ferenc, editors, *CSMR*, pages 319–328. IEEE, 2012.

[56] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 121–130, New York, NY, USA, 2009. ACM.

[57] C. Bird, A. Gourley, and P. Devanbu. Detecting patch submission and acceptance in oss projects. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 26, Washington, DC, USA, 2007. IEEE Computer Society.

[58] C. Bird, T. Menzies, and T. Zimmermann. *The Art and Science of Analyzing Software Data*. Morgan Kaufmann, 2015.

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

[60] C. Bird and T. Zimmermann. Assessing the value of branches with what-if analysis. In *Proc. of the ACM SIGSOFT 20th intl. symp. on the Foundations of Software Engineering (FSE)*, pages 45:1–45:11, 2012.

[61] C. Bird and T. Zimmermann. Assessing the value of branches with what-if analysis. In *Proceedings of the 20th International Symposium on Foundations of Software Engineering (FSE 2012)*. Association for Computing Machinery, Inc., November 2012.

[62] G. Booch. *Object Oriented Design: With Applications*. The Benjamin/Cummings Series in Ada and Software Engineering. Benjamin/Cummings Pub., 1991.

[63] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE11)*, pages 168–178, Szeged, Hungary, September 2011.

[64] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *Proc. of Foundations of Software Engineering (FSE)*, pages 168–178, 2011.

[65] J. Corbet. How to participate in the linux community. `http://ldn.linuxfoundation.org/book/how-participate-linux-community`, July 2008.

[66] J. Corbet, G. Kroah-Hartman, and A. McPherson. Linux kernel development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it. `http://go.linuxfoundation.org/who-writes-linux-2012`, April 2012.

[67] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Commun. ACM*, 31(11):1268–1287, Nov. 1988.

[68] C. R. B. de Souza, D. Redmiles, and P. Dourish. "breaking the code", moving between private and public work in collaborative software development. In *Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work*, GROUP '03, pages 105–114, New York, NY, USA, 2003. ACM.

[69] A. Dearle. Software deployment, past, present and future. In *2007 Future of Software Engineering*, FOSE '07, pages 269–284, Washington, DC, USA, 2007. IEEE Computer Society.

[70] T. Delaet, W. Joosen, and B. Van Brabant. A survey of system configuration tools. In *LISA*, 2010.

[71] A. Diklic. Unit tests for chef cookbooks with chefspec. `https://semaphoreci.com/community/tutorials/unit-tests-for-chef-cookbooks-with-chefspec`.

[72] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proc. of the 29th Intl. Conf. on Software Engineering (ICSE)*, pages 158–167, 2007.

[73] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 235–245, New York, NY, USA, 2014. ACM.

[74] R. Ellis. `http://www.spinics.net/lists/`, 2012. Last accessed in January 2012.

[75] J. Estublier and S. Garcia. Process model and awareness in scm. In *Proceedings of the 12th International Workshop on Software Configuration Management*, SCM '05, pages 59–74, New York, NY, USA, 2005. ACM.

[76] S. I. Feldman. Make-a program for maintaining computer programs. *Softw., Pract. Exper.*, 9(4):255–65, 1979.

[77] L. Fernández-Sanz and S. Misra. Influence of human factors in software quality and productivity. In *Proceedings of the 2011 International Conference on Computational Science and Its Applications - Volume Part V*, ICCSA'11, pages 257–269, Berlin, Heidelberg, 2011. Springer-Verlag.

[78] F. Fotrousi, S. A. Fricker, M. Fiedler, and F. Le-Gall. *Software Business. Towards Continuous Value Delivery: 5th International Conference, ICSOB 2014, Paphos, Cyprus, June 16-18, 2014. Proceedings*, chapter KPIs for Software Ecosystems: A Systematic Mapping Study, pages 194–211. Springer International Publishing, Cham, 2014.

[79] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, page 190–198, Washington, DC, USA, 1998. IEEE Computer Society.

[80] E. Giger, M. Pinzger, and H. Gall. Predicting the fix time of bugs. In *Proc. of the 2nd intl. workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 52–56, 2010.

[81] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grunbacher, and G. Antoniol. The quest for ubiquity: A roadmap for software and systems traceability research. *2013 21st IEEE International Requirements Engineering Conference (RE)*, pages 71–80, 2012.

[82] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grünbacher, A. Dekhtyar, G. Antoniol, and J. Maletic. *The Grand Challenge of Traceability (v1.0)*, pages 343–412. Springer-Verlag London Limited, 2012.

[83] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 156–166, Piscataway, NJ, USA, 2012. IEEE Press.

[84] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Proc. of the 32nd ACM/IEEE Intl. Conf. on Software Engineering (ICSE) - Volume 1*, pages 495–504, 2010.

[85] R. Hardt and E. V. Munson. Ant build maintenance with formiga. In *Proceedings of the 1st International Workshop on Release Engineering*, RELENG '13, pages 13–16, Piscataway, NJ, USA, 2013. IEEE Press.

[86] A. E. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57, Sept 2008.

[87] A. E. Hassan and T. Xie. Software intelligence: The future of mining software engineering data. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 161–166, New York, NY, USA, 2010. ACM.

[88] H. Hemmati, S. Nadi, O. Baysal, O. Kononenko, W. Wang, R. Holmes, and M. W. Godfrey. The msr cookbook: Mining a decade of research. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 343–352, Piscataway, NJ, USA, 2013. IEEE Press.

[89] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 483–493, Piscataway, NJ, USA, 2015. IEEE Press.

[90] K. Herzig and N. Nagappan. The impact of test ownership and team structure on the reliability and effectiveness of quality test runs. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 2:1–2:10, New York, NY, USA, 2014. ACM.

[91] Herzig, Kim and Nagappan, Nachiappan. Empirically Detecting False Test Alarms Using Association Rules. In *Companion Proceedings of the 37th International Conference on Software Engineering*, May 2015.

[92] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[93] A. Hindle, M. W. Godfrey, and R. C. Holt. Release pattern discovery: A case study of database systems. In *Proc. of the 23rd IEEE International Conference on Software Maintenance (ICSM)*, pages 285–294, 2007.

[94] http://bits.blogs.nytimes.com/2014/04/09/qa-on-heartbleed-a-flaw-missed-by-the masses/. Q. and a. on heartbleed: A flaw missed by the masses.

[95] https://code.google.com/p/gerrit/. Gerrit code review.

[96] http://www.itl.nist.gov/div898/handbook/. Nist/sematech e-handbook of statistical methods, 2010.

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

[98] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam. Testing idempotence for infrastructure as code. In *Middleware 2013*, pages 368–388. Springer, 2013.

[99] Y. Jiang and B. Adams. Co-evolution of infrastructure and source code – an empirical study. In *Proceedings of the 12th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 45–55, Florence, Italy, May 2015.

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

[101] P. Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, Code Snippets*, 28(1):1–9, October 2008.

[102] S. Karus and M. Dumas. Code churn estimation using organisational and code metrics: An experimental comparison. *Information & Software Technology*, 54(2):203–211, 2012.

[103] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams. 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*, MSR '12, pages 179–188, Piscataway, NJ, USA, 2012. IEEE Press.

[104] M. Kim and D. Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *MSR*, pages 1–5. ACM, 2005.

[105] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 81–90, 2006.

[106] S. Kirbas, A. Sen, B. Caglayan, A. Bener, and R. Mahmutogullari. The effect of evolutionary coupling on software defects: An industrial case study on a legacy system. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 6:1–6:7, New York, NY, USA, 2014. ACM.

[107] O. Kononenko, O. Baysal, and M. W. Godfrey. Code review quality: how developers see it. In *ICSE*, Austin, TX, USA, May 2016.

[108] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey. Investigating code review quality: Do people and participation matter? In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 111–120, Sept 2015.

[109] G. Kroah-Hartman. Android and the linux kernel community. `http://www.kroah.com/log/linux/android-kernel-problems.html`, Feb 2010.

[110] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *Proc. of the 29th Intl. Conf. on Software Engineering (ICSE)*, pages 106–115, 2007.

[111] Z. Lubsen, A. Zaidman, and M. Pinzger. Studying co-evolution of production and test code using association rule mining. In M. W. Godfrey and J. Whitehead, editors, *Proceedings of the 6th Working Conference on Mining Software Repositories (MSR 2009)*, pages 151–154, Washington, DC, USA, 2009. IEEE Computer Society.

[112] M. V. Mäntylä, B. Adams, F. Khomh, E. Engström, and K. Petersen. On rapid releases and software testing: a case study and a semi-systematic literature review. *Empirical Software Engineering*, 20(5):1384–1425, 2015.

[113] S. Mcintosh, B. Adams, and A. E. Hassan. The evolution of java build systems. *Empirical Software Engineering*, 17(4-5):578–608, Aug. 2012.

[114] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan. An empirical study of build maintenance effort. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 141–150, 2011.

[115] S. Mcintosh, Y. Kamei, B. Adams, and A. E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th IEEE Working Conference on Mining Software Repositories (MSR)*, Hyderabad, India, May 2014.

[116] A. Meneely and L. Williams. Secure open source collaboration: An empirical study of linus' law. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 453–462, New York, NY, USA, 2009. ACM.

[117] M. B. Miles and A. M. Huberman. *Qualitative data analysis : an expanded sourcebook*. Thousand Oaks, Calif. : Sage Publications, 2nd ed edition, 1994. Includes indexes.

[118] A. Mills. Why i quit: kernel developer con kolivas. `http://apcmag.com/why_i_quit_kernel_developer_con_kolivas.htm`, July 2007.

[119] A. Mockus. Organizational volatility and its effects on software defects. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 117–126, New York, NY, USA, 2010. ACM.

[120] A. Mockus, R. T. Fielding, and J. Herbsleb. A case study of open source software development: the apache server. In *Proc. of the 22nd Intl. Conf. on Software Engineering (ICSE)*, pages 263–272, 2000.

[121] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[122] K. Morris. *Infrastructure as Code -Managing Servers in the Cloud*. O'Reilly Media, 1st edition, 2016.

[123] I. Moura, G. Pinto, F. Ebert, and F. Castor. Mining energy-aware commits. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 56–67, May 2015.

[124] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. Association for Computing Machinery, Inc., May 2005.

[125] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: An empirical case study. Technical Report MSR-TR-2008-11, Microsoft Research, January 2008.

[126] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 315–324, New York, NY, USA, 2010. ACM.

[127] D. Parmenter. *Key Performance Indicators (KPI): Developing, Implementing, and Using Winning KPIs*. Wiley, 2010.

[128] D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large-scale software development: An observational case study. *ACM Trans. Softw. Eng. Methodol.*, 10(3):308–337, July 2001.

[129] H. Pickens. http://linux.slashdot.org/story/13/10/09/1551240/the-linux-backdoor-attempt-of-2003. The Linux Backdoor Attempt of 2003.

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

[131] F. Rahman and P. Devanbu. Ownership, experience and defects: A fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 491–500, New York, NY, USA, 2011. ACM.

[132] M. T. Rahman, L.-P. Querel, P. C. Rigby, and B. Adams. Feature toggles: A case study and survey. In *Proceedings of the 13th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 201–211, Austin, TX, May 2016.

[133] S. Rao and A. Kak. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 43–52, New York, NY, USA, 2011. ACM.

[134] P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 202–212, New York, NY, USA, 2013. ACM.

[135] P. C. Rigby and D. M. German. A preliminary examination of code review processes in open source projects. Technical Report DCS-305-IR, University of Victoria, January 2006.

[136] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: a case study of the apache server. In *Proc. of the 30th Intl. Conf. on Software Engineering (ICSE)*, pages 541–550, 2008.

[137] P. N. Robillard. The role of knowledge in software development. *Commun. ACM*, 42(1):87–92, 1999.

[138] G. Robles, J. M. Gonzalez-Barahona, and J. J. Merelo. Beyond source code: the importance of other artifacts in software development (a case study). *Journal of Systems and Software*, 79(9):1233–1248, 2006.

[139] P. Rodríguez, A. Haghighatkhah, L. E. Lwakatare, S. Teppola, T. Suomalainen, J. Eskeli, T. Karvonen, P. Kuvaja, J. M. Verner, and M. Oivo. Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*, pages –, 2016.

[140] P. Rotella and S. Chulani. Implementing quality metrics and goals at the corporate level. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 113–122, New York, NY, USA, 2011. ACM.

[141] G. Ruhe. *Product Release Planning - Methods, Tools and Applications*. CRC Press, 2010.

[142] A. Schaefer, M. Reichenbach, and D. Fey. Continuous integration and automation for devops. In *IAENG Transactions on Engineering Technologies*, pages 345–358. Springer, 2013.

[143] M. Serrano Zanetti. The co-evolution of socio-technical structures in sustainable software development: Lessons from the open source software communities. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1587–1590, Piscataway, NJ, USA, 2012. IEEE Press.

[144] T. Sharma, M. Fragkoulis, and D. Spinellis. Does your configuration code smell? In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 189–200, New York, NY, USA, 2016. ACM.

[145] E. Shihab, B. Adams, A. E. Hassan, and Z. M. Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 62:1–62:11, Research Triangle Park, NC, US, November 2012.

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

[147] M. Shridhar, B. Adams, and F. Khomh. A qualitative analysis of software build system changes and build ownership styles. In *Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Torino, Italy, September 2014.

[148] D. Spinellis. Don't install software by hand. *Software, IEEE*, 29(4):86–87, 2012.

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

[150] M. Taylor and S. Vargo. *Learning Chef-A Guide To Configuration Management And Automation*. 1st edition, 2014.

[151] S. A. Thomas, S. F. Hurley, and D. J. Barnes. Looking for the human factors in software quality management. In *Proceedings of the 1996 International Conference on Software Engineering: Education and Practice (SE:EP '96)*, SEEP '96, pages 474–, Washington, DC, USA, 1996. IEEE Computer Society.

[152] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 386–396, Piscataway, NJ, USA, 2012. IEEE Press.

[153] L. Torvalds. git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git, 2012. Last accessed in January 2012.

[154] J. Turnbull and J. McCune. *Pro Puppet*. Paul Manning, 1st edition, 2011.

[155] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 805–816, New York, NY, USA, 2015. ACM.

[156] P. Weissgerber, D. Neu, and S. Diehl. Small patches get in! In *Proc. of the intl. working conf. on Mining Software Repositories (MSR)*, pages 67–76, 2008.

[157] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Softw. Engg.*, 13(5):539–559, Oct. 2008.

[158] Wikipedia. http://en.wikipedia.org/wiki/f1_score, 2009.

[159] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

[160] C. K. Woodruff. Data processing people - are they really different? *Information & Management*, 3(4):133–139, 1980.

[161] T. Xie, K. Taneja, S. Kale, and D. Marinov. Towards a framework for differential unit testing of object-oriented programs. In *Proceedings of the Second International Workshop on Automation of Software Test*, AST '07, pages 5–, Washington, DC, USA, 2007. IEEE Computer Society.

[162] R. K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, 3 edition, 2002.

[163] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, Mar. 2012.

[164] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan. An empirical study on factors impacting bug fixing time. In *Proc. of the 19th Working Conf. on Reverse Engineering (WCRE)*, pages 225–234, 2012.

[165] T. Zimmermann. Mining workspace updates in cvs. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 11–14, Washington, DC, USA, 2007. IEEE Computer Society.