POLYPUBLIE

On the Cost and Value of Build Automation

Doriane Olewicki

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

Date	2020
Type	Mémoire ou thèse / Dissertation or Thesis Mémoire de maîtrise/Master's thesis
URL de PolyPublie PolyPublie URL	https://publications.polymtl.ca/5388/
Conditions d'utilisation Terms of use	© Tous droits réservés / All rights reserved
Référence Citation	Olewicki, D. (2020). On the Cost and Value of Build Automation [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/5388/

Détails de la thèse
Thesis details

Directeurs de recherche Advisors	Bram Adams, & Kim Mens
Programme Program	Génie informatique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal
https://publications.polymtl.ca

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

On the Cost and Value of Build Automation

DORIANE OLEWICKI
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise es sciences appliquées*
Génie informatique

Août 2020

© Doriane Olewicki, 2020.

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

On the Cost and Value of Build Automation

présenté par **Doriane OLEWICKI**
en vue de l'obtention du diplôme de *Maîtrise es sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Ettore MERLO, président
Bram ADAMS, membre et directeur de recherche
Kim MENS, membre et codirecteur de recherche
Foutse KHOMH, membre

DEDICATION

To Moeke and Vake,
who continue watching on me.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my advisor, Bram Adams, for his wisdom and his help all through my Master's. It was an honor working with you and taking part in exciting projects such as the Mozilla build activity research and the brown build project at Ubisoft. Being part of the MCIS lab has been a great experience. Your help and ideas inspired me and I am looking forward to continue working with you in the future.

Secondly, I am grateful to my co-advisor, Kim Mens, who kept interest in my projects and agreed to collaborate from UCLouvain in Belgium. Since I'm doing a double degree with UCLouvain and Polytechnique Montréal, it was great having one of my teachers from Belgium still following my path through my Master's and helping me out.

Then, I would like to thank my jury for accepting to judge my Master's defense. Starting with Ettore Merlo, it was great getting to follow your class and work with you on LOG3210. I enjoyed working with you and I am glad to show you the results of my research. Thanks, Foutse Khomh for taking part in my jury too. I also got the chance to follow one of your classes and to talk to you through the lab. I am happy to be able to show you what I did during Master's.

Next, I would like to thank Mathieu Nayrolles who worked with me on the Brown Build project at Ubisoft. Working with you is inspiring and I am glad to continue working with you in the following years. Thank you also to La Forge and Clever, the teams who made Ubisoft for me a nice place to work. Also, I am grateful to Ubisoft and Mitacs for the funding of the Brown Build project.

During my Master's, I got the chance to write and submit a paper to ICSME 2020 about my work on Mozilla's CI system. I would like to thank my co-authors, Noureddine Kerzazi, Kyle Lahnakoski and Ellis E. Eghan, for their collaboration and help through the research and writing process. Especially, thank you to Ellis who reviewed my thesis as well.

"Merci" Papa and "dank je" Mama for everything you did for me, for your support in my studies and all my decisions. Thanks to Marine for reviewing my thesis and for being a great sister. Thanks to my all family, my Querbes roommates, my Polytechnique lab friends and my friends in Montréal and Belgium. Last but not least, thanks to my partner, Timothée, for being the most supportive, to be with me in Montréal and for always showing how proud of me you are.

RÉSUMÉ

L'Intégration Continue (CI) est un processus de développement par lequel des changements de code d'un projet sont régulièrement testés et validés, souvent de manière quotidienne, afin de pouvoir mettre à jour plus régulièrement un produit. Ce système est de plus en plus utilisé notamment dans de grosses compagnies qui doivent gérer la participation de nombreux collaborateurs à un projet/produit commun. La validation du maintien de la qualité logicielle d'un produit est primordiale pour que le produit puisse être employé en continu par ses utilisateurs. Afin de faciliter cette intégration continue, les gestionnaires CI font souvent appel à des systèmes de builds automatisés: un build est une vérification de code via une compilation et un/des test(s) dans des conditions variées (différentes plateformes, versions, etc.). La gestion automatisée de ces builds permet aux systèmes de tester la validation de code à chaque arrivée de changements de code.

Cependant, pour des gros projets avec beaucoup de contributions, le nombre de builds peut très vite exploser. Les ressources doivent être exploitées de manière pertinente afin d'assurer la qualité logicielle au plus vite. De plus, générer des builds n'est pas gratuit: des ressources sont exploités, que ce soit le temps de calcul qui met en attente l'intégration, les machines locales au projet utilisées pour le build ou les ressources cloud. Ce coût est rarement abordé dans la littérature, mais ne doit pas être négligé.

Via cette thèse, nous cherchons à prouver que le coût de l'automation de build est non-négligeable et qu'il est possible de mettre en place des outils de réduction de l'activité de builds afin de réduire ce coût.

Dans le cadre de ma maitrise, j'ai eu l'occasion d'exploiter la gestion automatisée de builds dans deux projets : le premier concernant le projet libre d'accès Mozilla Firefox et le deuxième concernant des projets internes à Ubisoft Montréal.

La première étude que j'ai eu l'occasion de mener est une étude quantitative de plus de 50 millions de builds jobs dans le système CI de Mozilla au cours de 15 releases, soit 20 mois de développement. L'objectif de cette analyse est d'évaluer la force de builds générée à Mozilla ainsi qu'estimer le coût monétaire de la gestion automatisée de builds.

Le système CI de Mozilla, dans un souci de réduction de l'activité de build, rassemble plusieurs commits (changements de code) dans un "push", sur lequel une série de builds sont exécutés. Ceci permet de réduire le nombre de builds générés vu que la série de builds est

partagée par plusieurs commits; cependant, en cas d'erreur, il est plus difficile d'identifier le commit responsable. Mozilla utilise aussi un système de branches hiérarchiques par lesquelles doivent passer chaque changement de code lors de son intégration: chacune de ces branches ont une gestion différente de la taille des pushs et du nombre et type de builds qui sont déclenchés.

Nous nous sommes principalement concentrés sur le projet Mozilla Firefox dans cette analyse, qui est le plus gros projet de Mozilla, ainsi que sur quatre branches (Try, Mozilla-inbound, Mozilla-central et Mozilla-release) recommandées par un de nos contacts de Mozilla.

Dans un premier temps, nous avions rassemblé les build jobs de toutes les branches de Mozilla, ce qui nous a permis d'identifier que nos quatre branches d'intérêts couvrent plus de 50% des builds à Mozilla ainsi que 70% du temps de build.

Ensuite, nous avons observé l'activité par semaine de builds, de pushes et de commits. Nous avons ainsi pu voir que le nombre de builds par semaine augmente à travers le temps sur les différentes branches étudiées, alors que le nombre de commits par semaine réduit.

Nous introduisons alors le concept de *Effort Multiplier* (EM) qui est le nombre de builds par commit: vu que les commits sont regroupés dans des pushes, la responsabilité de chaque build est partagée entre les commits du même push. Nous calculons donc le EM en divisant le nombre de builds par le nombre de commits pour chaque push. Pour chaque branche étudiée nous avons observé que l'EM augmente au cours du temps. Ceci montre que malgré les efforts de Mozilla pour réduire la génération de builds, ceux-ci ne sont pas suffisants pour limiter l'augmentation de l'activité de builds.

Pour finir ce projet, nous voulions enfin donner une valeur monétaire à la génération de build. Nous avons estimé le coût des builds en heures machines par rapport aux coût d'allocation de machine dans le cloud via les services AWS On-Demande (borne supérieure) et Azure réservé sur trois ans (borne inférieure). Ceci nous a permis d'estimer que le coût médian de chaque commit du projet Firefox augmente au cours du temps de 19-45 USD lors de la release 36 à 79-125 USD lors de la release 50. De plus, moins de 10% du coût total est investi dans des builds apportant vraiment une information aux développeurs : des builds ayant échoués (indiquant donc une erreur dans les changements de code) ou les builds fixant les échecs de builds (indiquant alors qu'il n'y a plus d'erreur). Cela indique donc qu'il y a une large marge de réduction possible, que ce soit en sélectionnant ou en ordonnant les builds dans un sens logique.

La deuxième étude est une proposition de réduction de l'activité de builds, menée sur six projets internes à Ubisoft Montréal. Les développeurs de ces projets ont amené à notre

attention la présence de builds non-fiables dans leur système CI. En effet, certains builds vont parfois montrer des comportements incohérents et changer de résultat s'ils sont reproduits : ces builds sont identifiés comme des builds bruns (aussi: "flaky" builds). L'échec d'un build brun est donc dû non pas au changement de code mais au build lui-même (problème hardware, teste non fiable/flaky, etc.). La présence de ce genre de build a pour conséquence la perte de confiance des développeurs vis-à-vis du système CI : lorsqu'ils rencontrent un build échoué, les développeurs auront tendance à rerun le build avant de vérifier son changement de code.

Le taux de builds bruns dans les projets étudiés à Ubisoft varient entre 5 et 58% selon le projet. Cela confirme le besoin d'un détecteur de builds bruns pour ces projets afin de réduire l'activité de build et améliorer le système CI.

Dans le cadre de ce projet, nous avons développé un algorithme de détection de builds bruns langage indépendant et projet-agnostique. Le modèle est basé sur l'extraction de vocabulaire des traces de builds, afin de classifier les builds comme bruns ou non selon la ressemblance avec les builds du ensemble d'entrainement défini par projet. Les hyper-paramètres sont optimisés selon une optimisation locale (par projet) et une optimisation globale (même hyper-paramêtres pour tous les projets). L'optimisation local montre des performances encourageantes, atteignant en moyenne une exactitude de 85%, une précision de 55% et un rappel de 64%. La précision et le rappel sont assez bas comparés à l'exactitude, ce qui est dû au déséquilibre important du nombre de builds bruns par rapport aux builds non-bruns / vrai échec. L'optimisation global, quant à elle montre des performances moyennes similaires (exactitude à 82%, précision à 46% et rappel à 64%), indiquant que l'utilisation d'hyper-paramètres communs à tous les projets est possible.

Les performances de l'algorithme sont comparées par rapport à trois algorithmes de référence : *Random50* (prédit avec 50% de chance brun), *RandomB* (prédit avec B% de chance bruns, B étant le taux de builds bruns parmis les échecs de builds dans le training set) et *Always-Brown* (prédit avec 100% de chance brun). Les performances de ces références nous montre que notre algorithme est 30% plus efficace que la référence *Random50* et la précision près de deux fois plus grande que les trois références. Cela nous permet d'établir que notre algorithme a des performances représentatives et suffisamment bonnes pour améliorer le système CI en le rendant plus fiable.

En conclusion, nous avons montré que l'automation de build à un coût qui augmente au fil du temps selon l'avancement du projet, estimant ce prix pour Mozilla Firefox en moyenne à 19-45 USD par commit à la release 36, augmentant jusqu'à 79-125 USD par commit à la release 50. De plus, nous avons montré qu'il est possible de réduire l'activité de build,

en exploitant le cas des builds bruns par exemple. Pour les projects d'Ubisoft, nous avons détecté les builds bruns avec une exactitude moyenne de 82%, une précision de 46% et un rappel de 64%. Détecter et corriger ces builds permettent d'éviter la redondance de builds due au rerun manuel et augmente la fiabilité du système de builds.

ABSTRACT

Continuous Integration (CI) has become the core infrastructure of modern software organizations, enabling them to update projects more regularly (i.e., daily) while keeping the quality of the code valid and relevant. However, with more than 100,000 automated build jobs triggered each day by, for example, the Mozilla CI, build automation comes with a financial and time cost that should not be neglected.

Through this thesis, we aim at validating the hypothesis that the cost of build automation is not negligible and that the build activity could be reduced. This thesis presents the results of a quantitative study of the CI over the course of 20 months, analyzing over 54 million Mozilla build jobs to identify the cost of build automation triggered for each code change as well as a build reduction proposition, focusing on the case of brown builds at Ubisoft.

We find that the median build cost of a commit on the Mozilla Firefox project increases from 19-45 USD for release 36 to 79-125 USD for release 50. Less than 10% of that cost is incurred for builds that identified failures or validated a build fix, showing that the cost of automation could be significantly reduced by applying build selection or prioritization techniques on the other 90% of the build cost.

Furthermore, some builds have inconsistent results, i.e., not every failure is a real failure. Those unreliable builds are identified as brown builds. As a result, developers lose trust in the CI system and manually rerun failing builds without correcting potential errors in the code changes they pushed into their project. Rerunning automatically brown builds or correcting the brownness would improve the CI system and reduce build activity by improving the reliability of the system. This thesis introduces trained and validated models that provided early detection of brown build results with a mean accuracy of 84%, a mean precision of 46% and a mean recall of 64%.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

RÉSUMÉ v

ABSTRACT ix

TABLE OF CONTENTS x

LIST OF TABLES xii

LIST OF FIGURES xiii

LIST OF SYMBOLS AND ACRONYMS xiv

LIST OF APPENDICES xv

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 LITERATURE REVIEW 4
- 2.1 Continuous Integration 4
- 2.2 Build Failures 5
- 2.3 Test Prioritization and Selection 7
- 2.4 Brown Builds 7

CHAPTER 3 THE COST OF BUILD AUTOMATION AT MOZILLA 9
- 3.1 Introduction 9
- 3.2 Case study setup 10
  - 3.2.1 Research Questions 10
  - 3.2.2 Data Sources 11
  - 3.2.3 Preprocessing and Filtering 14
- 3.3 Results 15
- 3.4 Discussion 26
  - 3.4.1 Build cost per commit 26
  - 3.4.2 Total cost of builds per week 27

3.4.3 Fail-fix cost . . . 27
3.5 Threats to Validity . . . 28
3.6 Conclusion . . . 29

CHAPTER 4 BROWN BUILDS, A BUILD REDUCTION SOLUTION . . . 30
4.1 Introduction . . . 30
4.2 Brown-Detector Algorithm . . . 31
4.2.1 Feature extraction . . . 32
4.2.2 Vectorization . . . 33
4.2.3 Classification . . . 37
4.3 Case study setup . . . 39
4.3.1 Research Questions . . . 39
4.3.2 Data Sources . . . 39
4.3.3 Scraping . . . 41
4.3.4 Brown build identification . . . 42
4.3.5 Validation approach . . . 42
4.4 Results . . . 46
4.5 Discussion . . . 51
4.6 Threats to Validity . . . 52
4.7 Conclusion . . . 52

CHAPTER 5 CONCLUSION . . . 53
5.1 Summary of Works . . . 53
5.2 Future Research . . . 54

REFERENCES . . . 56

APPENDICES . . . 61

LIST OF TABLES

Table 3.1	Overview of the Meta Information Extracted.	12
Table 3.2	Number of commits gathered from Mercurial on each branch of interest, from 2014-12-11 to 2016-10-30.	12
Table 3.3	Size of the data before and after preprocessing.	13
Table 3.4	Metrics on the complete data-set by branch. The table is split in two parts: the first regarding build compilation only and the second on build test.	16
Table 3.5	Medians of the different measures by branch	22
Table 3.6	Pricing [USD] of build computation time per hour, depending on the platform and the type of build (compilation or test).	23
Table 4.1	Information on the projects studied.	40
Table 4.2	Scrapped data for each build job on each project.	41
Table 4.3	Baselines. (With BFR the brown rate among the failed build jobs by project)	45
Table 4.4	Metrics on the studied projects.	46
Table 4.5	Model performance.	47
Table 4.6	Hyper-parameters chosen through the optimization of F1-score.	49
Table D.1	Example of application of the six rules of vocabulary reduction.	70
Table E.1	Example of application of TF-IDF.	71

LIST OF FIGURES

Figure 3.1	Code change flow between the studied branches.	15
Figure 3.2	Summary of Firefox's build activity by week of releases 36 to 50.	19
Figure 3.3	Loess regression of the median values on each branch over weeks of releases 36 to 50.	21
Figure 3.4	Loess regression of the median values for (a) and total for (b) and (c) with lower-bound (Azure approach) and upper-bound (AWS On-Demand) on each branch over weeks of releases 36 to 50.	25
Figure 4.1	Iterative vectorization algorithm	35
Figure 4.2	Two-step model training	38
Figure 4.3	Prediction with two models	38
Figure 4.4	Optimization of the hyper-parameters.	48
Figure 4.5	Comparison of the mean F1-score performance by hyper-parameters.	50
Figure A.1	Number of builds by push (#build/push) on each branch on each branch over weeks of releases 36 to 50.	61
Figure A.2	Number of commits by push (#commit/push) on each branch on each branch over weeks of releases 36 to 50.	62
Figure A.3	Number of builds by commit (EM) on each branch on each branch over weeks of releases 36 to 50.	63
Figure B.1	Price (USD) by commit on each branch based on AWS' pricing.	64
Figure B.2	Fix-fail price (USD) by commit on each branch based on AWS' pricing.	65
Figure B.3	Total cost (USD) of builds by week on each branch based on AWS' pricing.	66
Figure C.1	Price (USD) by commit on each branch based on Azure's pricing.	67
Figure C.2	Fix-fail price (USD) by commit on each branch based on Azure's pricing.	68
Figure C.3	Total cost (USD) of builds by week on each branch based on Azure's pricing.	69

LIST OF SYMBOLS AND ACRONYMS

CI	Continuous Integration
R36	Release number 36 of Mozilla's development (we studied release 36 to 50)
EM	Effort Multiplier, defined as $EM = \frac{\#build}{\#commit}$ for each push
#build/push	Number of builds per push
#commit/push	Number of commits per push
AWS	Amazon Web Services
USD	United States Dollar
TF-IDF	Term Frequency - Inverse Document Frequency
Regex	Regular expression
BFR	Brown rate among the failed build jobs ($BFR = \frac{\#brownfailjob}{\#failjob}$)
TP	True Positive
FP	False Positive
TN	True Negative
FN	False Negative
Acc	Accuracy ($\frac{TP+TN}{TP+TN+FP+FN}$)
Pre	Precision ($\frac{TP}{TP+FP}$)
Rec	Recall ($\frac{TP}{TP+FN}$)
F1	F1-score ($\frac{2}{Pre\hat{}-1+Rec^{-1}}$)

LIST OF APPENDICES

Appendix A Complementary results for RQ2 61

Appendix B Complementary results for RQ3, AWS approach 64

Appendix C Complementary results for RQ3, Azure approach 67

Appendix D Example of vocabulary reduction for the Brown Build detection methodology. 70

Appendix E Example of TF-IDF application for the Brown Build detection methodology. 71

CHAPTER 1 INTRODUCTION

> "*Force Multiplier is a capability that, when added to and employed by a combat force, significantly increases the combat potential of that force and thus enhances the probability of successful mission.*"
>
> The Oxford Essential Dictionary of the U.S. Military

Build automation, which involves source code compilation and tests, plays a crucial role in contemporary software development organizations [1]. Increasingly, it is being viewed as a force multiplier of productivity and quality. This is because most modern development teams leverage build automation to support the agile practice of Continuous Integration (CI) [2,3], which encourages developers to merge their code changes into the main project several times a day. Before being merged, these code changes trigger builds that validate and guard the quality of the project codebase. Builds, in the context of this thesis, refer to both compilation and test processes.

If done correctly, build automation boosts team productivity in many ways: *(1)* ensuring that automated builds are consistent ; *(2)* improving product quality by executing automatically tests on one or more targeted platforms and settings; *(3)* reducing manual labor in the release pipeline by running a predefined pipeline of tasks that automatically package, test and deploy software to production.

However, a given code change is not only built once; it is rather built several times using multiple combinations of different environments (e.g., Linux, Windows) and library versions (e.g., x86-64). Although these combinations are required to ensure that the system will not fail on any platform users may work on, they lead to an inflation of builds [4,5]. Apart from taking more time to finish all builds, and hence slowing down build feedback to developers, having more builds also imply a higher financial cost, since builds do not come for free — the machine resources, money and time spent to generate builds cannot be ignored. Yet, only a back-of-the-napkin calculation by John O'Duinn [6] provides an approximate cost of 30.60 USD per push for build automation at Mozilla, where a push is a group of commit.

This thesis hypothesizes that build automation has a cost that increases over time and that efforts should be invested into reducing that cost.

Naturally, the build activity evolves over time, due to the natural growth in size of a project over its lifetime and the increasing maintenance that comes with this. However, although

such growth might lead to longer builds, other factors like the number of contributors, size of the test suite, number of supported configurations and environments, etc. all play a role as well. The point of this thesis is that not all of these factors lead to equally useful builds for developers.

Using two large-scale empirical studies on real build data from the open-source Mozilla projects and closed-source Ubisoft projects, this thesis aims at evaluating the real cost of build automation in terms of the number of build jobs being started, and the duration of those jobs, and the return on such an investment. Furthermore, we evaluate the extent to which the prediction of brown builds (unreliable builds that may fail due to factors other than code changes) can be used to reduce the build activity, and thus the cost of builds.

In Chapter 3, we identified Mozilla's build activity and we evaluated the cost of build automation on Mozilla's project. We examine the archive of ∼54 million build jobs from Mozilla, over the course of 20 months in order to study the activity and cost of these builds, as well as their usefulness in identifying build failures. We propose and measure the notion of Effort Multiplier (EM) of build automation, which is the number of builds triggered after each code change. Using EM, our research aims at evaluating the build cost in terms of machine computation time, both at the level of individual commits and aggregate pushes (a popular mechanism to counter build inflation).

The second study proposes and evaluates an approach for build reduction by detecting brown builds on closed-source projects at Ubisoft Montréal. Brown builds are unreliable builds that might change results if they were rerun: for instance, a failed build job might succeed if it was rerun. Those builds do not bring any interesting information to the developers since the error is due to build problems instead of the code change itself. In Chapter 4, we describe our brown build detection algorithm based on vocabulary extraction from the build jobs' traces and validate it on six projects of various sizes, programming languages, and purposes. The resulting model is language and project agnostic, making it easy to apply to any project that uses an automated build system.

The thesis makes the following contributions:

- A quantitative study of the evolution of build activity at Mozilla over a period of 20 months, comprising more than 50 million build jobs. We believe that the research community about CI usage will get more insights from such a large build system dataset, increasing awareness about the role of CI practices in the trenches.
- An empirical analysis of the EM ratio in a real-world build automation environment. This will allow us to observe how each commit impacts the build activity in a system

in which several commits are built together.

- A cost estimation of build automation based on the computation time and environment.
- A proposal and empirical evaluation of a language- and project-agnostic brown build detection algorithm to reduce the build activity and thus cost.

CHAPTER 2 LITERATURE REVIEW

In this chapter, some related work to my thesis is presented. The chapter is divided into several section, that refer to one or both of my projects (Mozilla build automation analysis and brown build detection at Ubisoft).

2.1 Continuous Integration

First, let us introduce the concept of Continuous Integration. A Continuous Integration (CI) server automatically re-builds and re-tests the source code of a project whenever a developer pushes a code change to her version control system, in order to detect faults and merge conflicts as soon as possible [2]. A typical CI pipeline like Jenkins or TravisCI consists of a sequence of build stages (e.g., "compile" followed by "test'), each composed of one or more parallel build jobs (e.g.,"compile" jobs on Linux and Windows). The behaviour of such a build job is specified via build scripts in a domain-specific language like GNU Make[1], Maven[2] or Gradle[3]. Such scripts typically transform source code into an executable program by invoking configuration tools, preprocessors, and compilers, they automate the execution of test harnesses and/or can even deploy the produced build artifacts [2, 7].

While, conceptually, a CI server is thought of as performing one build for each new code change, in practice its role is much more complex. First of all, the build dashboards of large open source organizations like Mozilla's treeherder[4] or OpenStack's zuul[5] show a multi-dimensional matrix that tries to summarize results for dozens of CI pipelines and build jobs, ranging from classic compilation and unit test execution to deployment or even static analysis.

Furthermore, such pipelines are run multiple times for a given code change, since within each build stage multiple jobs should be run to cover the major feature and environment configurations on which the code base is expected to run. For example, if a project has 10 features and should support 5 operating systems, ideally 10 x 5 build jobs should be scheduled in each build stage. Since a typical project has a much larger number of features, and its environment comprises not only different operating system versions, but also different devices, processor models, library dependencies, supporting databases and web servers, each code change potentially yields a combinatorial explosion of build jobs to run. Of course, if a

[1]GNU make: `https://www.gnu.org/software/make/`
[2]Maven: `https://maven.apache.org/`
[3]Gradle: `https://gradle.org/`
[4]Mozilla's treeherder: `https://treeherder.mozilla.org/`
[5]OpenStack's zuul: `https://zuul.openstack.org/`

build is deemed to fail, all builds would need to be repeated again for the proposed code fix.

While this phenomenon, which in earlier works was named "build inflation" [8], increases confidence in build results, it brings a number of major disadvantages as well. First of all, it increases the build infrastructure and energy cost for organizations. Google, for instance, performs 800k builds per day, which schedule 150M test runs [9]. Google's breakneck code velocity of one commit per second coupled with its linearly increasing test corpus implies a quadratically growing need in build resources that even Google is no longer able to handle. Similar to many other organizations, such as OpenStack, Google has adopted CI scheduling algorithms that group multiple code changes (e.g., all changes arriving within 45 minutes) within a "push" before starting a new build on the entire group, instead of executing separate builds for each code change. While successful builds at the group-level save many builds at the individual level, failures at group-level do require additional follow-up builds to determine the individual code changes responsible for the failures.

Vasilescu et al. [10] demonstrated that CI significantly improves the productivity of development teams through an empirical study of 246 projects from GitHub all of them using Travis CI. However, the authors did not quantify the benefits of such CI usage, such as an effective error detection.

Hilton et al. [11] studied the use of continuous integration (CI) in a large number of open source projects, pointing out several key questions related to the usage, adaptation costs, and benefits of CI. They show evidence that CI practice helps projects release more often. The developing interests on CI and the will to improve CI practices [2] motivate this thesis, especially Chapter 3 on CI and builds at Mozilla.

In this thesis, we analyze the build activity of the CI system at Mozilla, regarding the Mozilla Firefox project. From this analysis, we evaluate how the build automation impacts the build activity and the cost of using such a CI. Though the Ubisoft Brown Build project, we want to improve the CI system, by detecting brown builds, a problem that will be describes in a following section.

2.2 Build Failures

CI gives information to developers about the reliability of a code change going through integration: should a code change be accepted in the main source code and be forwarded to the user or should the code be rejected and corrected by a developer? Build failures are results that actually bring information to developers and prevent errors to happen at the user level. They have a real value in the CI since they detect problems before the code gets

pushed through to the users. In this section, we present other researches on build failures analysis.

Beller et al. [12] explored 2.6 million Java and Ruby builds of the Travis CI data-set and found out that testing is the main reason for build failures. In our work, we complement previous works by quantifying the Effort Multiplier provided by the build team at Mozilla, over the 53 millions of builds generated over 15 releases.

Seo et al. [13] quantitatively studied 26.6 million C and Java builds at Google to find root causes (patterns) of failures considering cross-language Java and C++ systems. They found a higher error rate for C++ (38.4%) than for Java (28.5%). Specifically, the authors identified the 25 most frequent kinds of error messages for Java and C++, which they categorized in: *Dependency*, *Type Mismatch*, *Syntax*, *Semantic*, and *Other*. Hochstein et Jiao [14] studied the build outcome for two projects and reported 11% of build failures.

Neitsch et al. [15] analyzed build outcomes for multi-language packages and found that the developers were unaware of certain build problems at least within their own environment. Authors addressed certain well-known anti-patterns for common build failures such as "*Incorrect Dependencies*" and "*Filename Collision*".

Kerzazi et al. [16] carried out a mixed study in an industrial context aimed at quantifying the impact of build breakage on software project development and categorizing factors leading to unsuccessful builds. They report a ratio of build breakage as high as 17.9% over 6 months and an estimated direct cost of build breakage as more than 336.18 manhours. Authors also found out that only 28% of the team members are aware of build breakage, taking on average 3 hours to be detected and typically an hour to be fixed.

Other studies have analyzed builds in the context of defect prediction using build results as an input for their prediction models. For instance, Kwan et al. [17] examined build failure from the perspective of a socio-technical dimension. They discovered that there is a relationship between socio-technical congruence and build success probability, but only for certain build types.

In this thesis, we look at failure as interesting build results, since it highlights potential errors in the code changes. In the context of the Mozilla build activity analysis, we estimate what part of the cost of build automation is attributed to failure and that is thus a valuable investment of the build automation system.

2.3 Test Prioritization and Selection

CI generates many builds that require a lot of machine resources, in particular regarding the computation time, and in human time, since the developers wait for the CI's feedback. Since most of the build activity is testing the code, techniques of test prioritization and selection could be applied to reduce the build computation time and thus cost. In this section, we present existing techniques that might be applied.

Time and cost-aware techniques have been used in previous empirical studies to select a subset of test suites that can be executed on a given time budget [18–21]. Herzig et al. [18] used a cost-aware approach that dynamically skips tests when the expected cost of running the test exceeds the expected cost of removing it, intending to accelerate the test process without sacrificing the product quality. They tested this algorithm by simulating build selection on historical test runs: Tests were marked as skipped or executed depending on the prediction of failure and the cost prediction of running the test. Authors found a reduction of 50% in test computation time, saving millions of dollars per year, while maintaining product quality.

Other empirical studies [19, 21] used time-aware techniques to assess the effect of time constraints on the cost and benefits of test prioritization techniques. Ghaleb et al. [22] studied the factors of long duration of continuous integration builds, from 67 GitHub projects, which might reduce the productivity of the contributors. Other studies assessed the effect of time constraints on the cost and benefits of prioritization techniques. Li and Boehm [23] assessed a value-based test prioritization strategy ranking tests by their risk exposure coverage and the relative cost of tests.

Our study on the Mozilla project aims to complement prior work on build automation by examining the build activity and cost by commit over time within a specific context and to motivate work on prioritization. To the best of our knowledge, this study is the first that evaluates value-based and cost-effectiveness based on the Mozilla context. Through the Ubisoft Brown Build project, we present and evaluate a build selection algorithm based on brown build detection.

2.4 Brown Builds

In this thesis, we study a build reduction approach. However, instead of following the test prioritization and selection techniques that exist, we focus on the brown build project, to improve the CI build automation system.

In fact, while the interpretation of build failures is the main challenge of CI users [24], the

large number of builds generated by build inflation make build failures harder to interpret. For example, Gallaba et al.'s analysis of 3.7 million GitHub build jobs [25] found that 12% of passing CI builds contain failing or skipped build jobs that were actively ignored by developers. Moreover, two out of three breakages occur for more than one CI run, meaning that they were ignored by developers at least once. Furthermore, 44% of the studied build failures were environment-dependent, i.e., only occurred for some environments.

Even worse than the presence of noise and build inflation due to different environments, is the ambiguity of build results caused by so-called brown (or flaky) build jobs. These are build jobs that fail inconsistently due to issues with asynchronous calls, multi-threading, or test order dependencies [26]. One can determine whether a build job failed or succeeded only by repeating such build jobs a sufficiently large number of times. Hence, in case of doubt, the most conservative reaction to a possibly real build failure is to lock the code contribution pipeline, basically preventing other teams from merging their contributions. If one would be able to know that the build is truly flaky, the build's failing outcome could be ignored and the pipeline could continue the integration process.

Since, in a typical organization, different software components are reused across different libraries and products, the (apparent) success of a new build tacitly is considered as a "go" signal for dependent products to adopt the new release of the component. Flaky builds would render such adoption premature and propagate flakiness to the adopters' build pipeline.

While certain build tools like Maven have support for rudimentary flaky test detection through build repetition, this is not efficient, nor accurate. For this reason, existing work in this area [26–29] has focused on empirically understanding the causes of flaky tests, as well as ways to detect such tests for specific programming languages. Ironically, there are often resources available to optimize and fix the build environment or source code responsible for flaky builds, yet those need to be briefed with concrete starting points, which currently are not known.

Brown builds can, thus, be due to flaky tests or environmental problems from the build settings. Approaching the brown build problem in a language dependent way is difficult, since builds often run using several languages.

This thesis proposes a language-independent solution to brown build detection, based on the vocabulary extraction of the build jobs' logs. This approach is evaluated on six closed-source projects at Ubisoft. With this brown build detection system, we hope to improve the CI system by preventing false negative build results and improve the build jobs them-self, so that developers rely more on the CI.

CHAPTER 3 THE COST OF BUILD AUTOMATION AT MOZILLA

This chapter presents the research on the build automation system at Mozilla. The purpose of this research is to evaluate the build activity and cost of the Mozilla Firefox project over a period of 20 months. Build activity here relates to the overall number of builds that are generated by the Mozilla pipeline in response to incoming source code changes, while cost estimation relates to the financial cost of build activity and the resources that are used to do the build automation. We do not consider the coding effort by developers and the build maintainers, nor the delay due to the pipeline integration.

Furthermore, this research aims to validate the hypothesis that the build activity increases over time and that an effort should be invested in reducing the cost. It should be noted that we do not present a solution yet; a potential solution is proposed in the next chapter. Other possible solutions will be briefly discussed in this chapter.

3.1 Introduction

Mozilla has a build process generating more than 100,000 build jobs per day. The practice of automation is used extensively at Mozilla for about 10 of its projects with large code-bases and a large community of contributors (∼3,000). Those builds are triggered to validate the quality of code changes across 43k build configurations over the studied period; these configurations consist of the type of build (compilation or test) and the platform choice (one of the 103 possible platform-version supported).

John O'Duinn roughly evaluated the cost of build automation at Mozilla back in 2013 [6]. For this, he computed a back-of-the-napkin cost over a month period and evaluated a push at 30 USD. His calculations were made with the assumption that in-house builds (i.e., builds not done in the cloud) had a zero cost; only the costs of the Linux builds run on AWS were considered.

In this study, we aim to extend John O'Duinn's cost evaluation. First of all, our studied period extends over about 20 months and 15 releases. Furthermore, we consider the cost of other platforms (MacOS and Windows), since in-house costs are not negligible (machine resources, maintenance, and energy consumption): we considered cloud computing cost as well for those platforms since in-house costs are difficult to estimate without inside private information. The cost evaluation gives an idea of the financial impact of using a build automation system over time (each commit's builds gave a cost) and highlights the need to improve the build

(compilation and test) selection. However, we do not consider the development cost (cost of the developer implementing the code changes), nor the integration delay: we only focus on the computation time and resource consumption of build automation. Apart from a longer studied period, our research provides in-depth analysis of four main branches of the Mozilla ecosystem, focusing on their largest project (Mozilla Firefox), analyzing the build activity at both the push and the commit level, and introducing the notion of Effort Multiplier.

3.2 Case study setup

The main goal of our chapter is to quantitatively analyze the build activity carried out by the build team at Mozilla, as well as its cost.

3.2.1 Research Questions

This chapter will answer the following research questions.

RQ0: *What is the overall distribution of build activity across Mozilla projects?*

Mozilla's build activity is distributed over several projects and branches used as a path to integration. Two types of builds are generated: compilation and test builds (both types do not overlap, test build jobs do not do compilation and vice-versa). With this question, we seek to evaluate the distribution of the builds across those types, as well as on the different integration branches in all Mozilla's projects. Mozilla works on several projects among which the most important and well-knows is Firefox. This question studies the number of builds triggered by the build automation system at Mozilla over time, as well as the development activity in terms of the number of code changes submitted over time. The next RQs will then focus on the largest Mozilla project (i.e., Firefox).

RQ1: *How does build activity evolve across time in Firefox?*

Code changes are often grouped and scheduled into "pushes" to optimize resources, running builds once for a sequence of commits instead of each commit. With this question, we focus on the weekly evolution of build activity on our branches of interest, regarding the number of code changes, the pushes, and builds.

RQ2: *How many builds are triggered per commit?*

With this question, we want to evaluate the Effort Multiplier, which measures highlight the build activity attributed to each commit (e.g. if 5 builds are triggered for a push of

2 commits, the EM is 2.5). This allows a developer to have a better idea of the impact of his changes on the build infrastructure as well as insights in terms of the expected value of the builds.

RQ3: *How much does a single commit in Firefox cost?*

In a large project, an effective build process is key to ensure quality throughout the pipeline of the release. However, builds do not come for free due to the cost of the infrastructure (servers, clouds, etc.) on which builds are run. We are interested in assessing the cost of every single build, triggered on a given branch, which includes building, packaging, testing, and so on. We also want to evaluate the return on investment (ROI), as the percentage of builds that actually brought information to the developer about potential faults in the code changes or about the success of a proposed fix (first successful build after a breakage). Those builds are the ones that provide information to developers and that helps improve the code quality.

3.2.2 Data Sources

Mozilla handles builds and automation using the Buildbot system[1]. Buildbot is a job scheduling system at its core. It queues jobs, executes the jobs when the required resources are available, and reports the results. Buildbot has one or more masters and a collection of builders (workers). The masters monitor source-code repositories for changes, coordinate the activities of the builders, and report results. The build system at Mozilla used 43k distinct Buildername (machines) in the studied period: the Buildername is an identifier that gives information about the different combination of operating system (i.e., Linux, MacOS, or Windows) with its specific version (103 different platform-version combinations), the type of build (test or compilation) and other specific build settings.

The data gathered covers the releases R36 to R50 (fifteen consecutive releases) from January 2015 to the end of October 2016. This data was graciously provided by the Mozilla build team. Each release is composed of five to eight weeks. The scraping of that data took a lot of effort and the size of those 15 releases data-set is about 18 GB for 54 millions of build jobs. To put these numbers in context, TravisTorrent data-set comprises 3.7 million builds [30], covering only one platform (Ubuntu).

We extract relevant information from the raw archive data, publicly exposed by Mozilla's build team in *.json* format [2]. For each build log in the *.json* file we retrieve information

[1]Buildbot: `http://buildbot.net/`

[2]ActiveData: `https://activedata.allizom.org/tools/`

Table 3.1 Overview of the Meta Information Extracted.

Fields	Description
builduid	a unique Id of 32 digits
starttime	timestamp indicating the start time
endtime	timestamp (duration)
buildername	one of the 42,676 Buildbot builders
branch	one of the 46 source code branches
platform	one of the 103 platforms supported
product	one of the 9 products
reason	reason for triggering the build
result	build status: from 0 to 6

described in Table 3.1. In the build data-set, the result of each build is represented by a number between 0 and 6: 0 indicates a successful build, while any other value refers to a specific type of failure or warning. All failures and warnings are categorized as failed builds in this study. Builds also contain information about the platform on which they were executed.

Each push (set of commits built together) is identified by the revision number of the last change-set in the group of commits, and each build is associated with a push's revision number (making it possible to identify which build was done on which push) and a specific branch. This parameter will be used to identify the activity generated by each push on each branch separately and to attribute build activity to the commits in the push.

We retrieved the commits from the Mercurial repository of Mozilla. We only gathered the data for a few branches of interest, which will be explained in the next section. The data was gathered from the 11th of December 2014 (a month before the release r36) to the 30th of November 2016. We gathered a total of 884,719 commits, detailed by branch in Table 3.2.

Table 3.2 Number of commits gathered from Mercurial on each branch of interest, from 2014-12-11 to 2016-10-30.

Branch	#Commit
Try	567390
Mozilla-inbound	101044
Mozilla-central	101053
Mozilla-release	115232
Total	884719

Table 3.3 provides the number of pushes and builds by release before the preprocessing of the next section. The median number of pushes per release is around 12k and the median number of builds around 3.5 million.

Table 3.3 Size of the data before and after preprocessing.

Release	#Weeks	Start date	End date	All products and branches			The 4 analyzed branches of Firefox			
				#Pushes	#Builds	%Fail	#Commit	#Pushes	#Builds	%Fail
r36	6	2015-01-12	2015-02-22	15,722	3,200,033	10%	42,358	7,355	1,189,488	8%
r37	5	2015-02-23	2015-03-29	13,983	2,969,695	10%	45,922	7,294	1,252,690	9%
r38	6	2015-03-30	2015-05-10	12,795	3,020,443	9%	58,650	8,609	1,370,163	7%
r39	7	2015-05-11	2015-06-28	13,769	3,177,828	10%	31,368	9,410	1,408,273	8%
r40	6	2015-06-29	2015-08-09	11,495	3,133,501	8%	54,211	8,231	1,460,616	9%
r41	6	2015-08-10	2015-09-20	12,186	3,326,959	9%	72,467	8,445	1,639,065	9%
r42	6	2015-09-21	2015-11-01	12,581	3,792,639	8%	54,828	8,271	1,843,981	9%
r43	6	2015-10-29	2015-12-13	11,898	3,603,840	8%	48,902	7,384	1,776,021	9%
r44	6	2015-12-14	2016-01-24	8,624	3,041,986	7%	39,662	6,293	1,568,187	8%
r45	6	2016-01-25	2016-03-06	10,670	3,761,726	8%	35,309	8,249	2,015,072	7%
r46	7	2016-03-07	2016-04-24	12,583	4,493,167	6%	50,353	9,908	2,626,738	5%
r47	6	2016-04-25	2016-06-05	10,824	3,804,329	7%	40,741	8,888	2,279,498	8%
r48	8	2016-06-06	2016-07-31	11,944	5,140,127	6%	41,076	9,245	2,782,949	7%
r49	7	2016-08-01	2016-09-18	12,187	4,372,210	6%	50,831	9,116	2,275,803	7%
r50	6	2016-09-19	2016-10-30	10,563	3,626,508	6%	56,093	7,656	1,873,667	7%
Median				11,898	3,603,840	8%	48,902	8,271	1,776,021	8%
Total				181,824	54,464,991	8%	722,771	124,354	27,362,211	8%

3.2.3 Preprocessing and Filtering

Except for RQ0, which reports on the overall distribution of the entire build activity across all Mozilla projects, the other RQs will focus on the Firefox project. We focus on the Firefox product because it is the biggest project at Mozilla, covering 80% of all build jobs of the Mozilla infrastructure.

Mozilla uses a hierarchical version control of repositories as "branches". Each branch has its own position on the integration path and all commits should pass through all branches to go to the release. A simplified representation of that hierarchy is represented in Figure 3.1. RQ0 will study the "by branch" activity. The other RQs will only study four branches (i.e., Try, Mozilla-inbound, Mozilla-central, and Mozilla-release) The selected branches are representative of the overall activity: as we will see, they cover 70% of the build computation time.

As advised by the build team at Mozilla, filtering on products and branches prevents noisy data from experimental branches or projects from altering our results. This choice will be further discussed in RQ0. On the other hand, the filtering also means that the calculated costs in RQ3 are lower than the overall cost of all branches and projects.

The chosen branches have a hierarchical behaviour regarding code changes integration path, as can be seen in Figure 3.1. Each branch is described as follows:

- **Try**: a branch on which any contributor can build and test code changes before submitting them for review. The builds are thus simply informative; they let the developers know if the changes made are good enough to be submitted, using the same build automation as will be used during code review.
- **Mozilla-inbound**: a post-review branch on which modifications are built and tested. This branch is one of the first steps into the integration of code changes. Those builds are considered more important because they assess the quality and reliability of changes in each environment as they enter the repository.
- **Mozilla-central**: a branch similar to Mozilla-inbound, but more downstream in the integration process, at an intermediate state between the beginning of integration (just after review) and the final release version of the code.
- **Mozilla-release**: the closest branch to release. This is the last step of code refinement toward the final product.

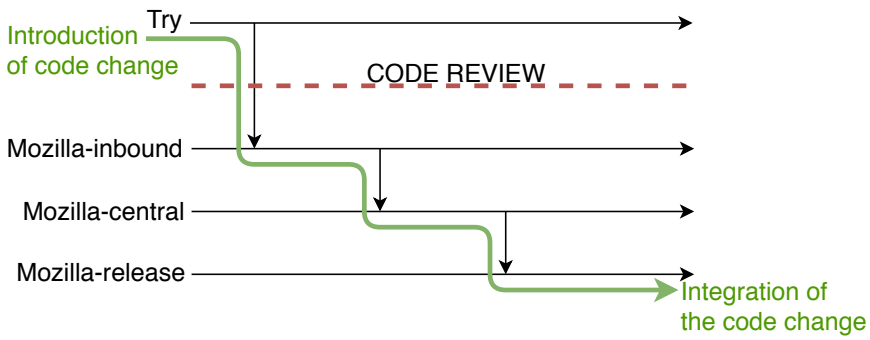


Figure 3.1 Code change flow between the studied branches.

3.3 Results

This section describes and analyzes the results achieved when answering our research questions.

RQ0.What is the overall distribution of build activity across Mozilla projects?

Motivation:

To evaluate the build activity across the different branches used in the hierarchical organization of Mozilla's source code integration, we gathered metrics in Table 3.4 on the build activity by branch and by type of build (compilation or test) before processing. Mozilla's build team recommended us to study four branches (Try, Mozilla-inbound, Mozilla-central, and Mozilla-release). In this question, we want to see how representative those branches are regarding the overall activity. Furthermore, we complement previous works by quantifying the build activity of the Mozilla system and evaluating the failure rate on the different branches.

Approach:

In Table 3.4, the rows are ordered according to the total number of builds run on the branches. #Buildername represents the number of different builders that were used over the period the data covers. #Builds the total number of builds that were run. The first Time column is the total duration of a build and the second column is the percentage of time spent on that type of build regarding the total time on the branch.

Results:

A median of 3.6 million build jobs are triggered per release. The number increases

Table 3.4 Metrics on the complete data-set by branch. The table is split in two parts: the first regarding build compilation only and the second on build test.

Branch	Compilation builds				Test builds			
	#Builder-name	#Build (Fail%)	Time[d h:m]	Time[%]	#Builder-name	#Build (Fail%)	Time[d h:m]	Time[%]
Try	54	677,755 (18%)	24794 10:53	5.42	3,154	18,565,904 (12%)	433067 05:29	94.58
Mozilla-inbound	37	439,729 (5%)	18249 16:10	5.30	2,962	14,614,138 (5%)	326113 03:26	94.70
Fx-team	35	136,111 (4%)	6894 13:50	5.29	2,959	5,900,484 (4%)	123350 20:54	94.71
Mozilla-central	38	50,642 (6%)	2999 22:36	4.22	3,087	3,154,593 (4%)	68046 12:05	95.78
Mozilla-aurora	19	30,504 (9%)	2260 07:00	4.28	2,794	2,310,867 (4%)	50537 14:44	95.72
Autoland	18	32,661 (5%)	1890 16:20	4.37	1,781	2,099,472 (4%)	41328 11:52	95.63
B2g-inbound	35	56,393 (3%)	3120 14:07	5.91	1,764	1,887,219 (5%)	49679 13:18	94.09
Mozilla-beta	23	22,971 (9%)	1758 05:16	5.58	2,202	1,371,143 (5%)	29734 21:32	94.42
Mozilla-release	23	4,244 (9%)	331 18:23	6.32	2,042	223,690 (5%)	4918 19:58	93.68
Other 37 branches	417	48,542 (12%)	2740 06:34	6.48	19,232	1,633,860 (8%)	39554 06:41	93.52

significantly over time, even though the number of pushes decreases over time, with a median of 12k per release.

Table 3.3 shows how the number of builds increases significantly (with alpha=0.01 and a large effect) over time from around 3 million to 4 million, while the number of pushes decreases (with alpha=0.01 and a medium effect) from 15k to 11k. Furthermore, only a median of 8% of those builds fail. Over time, the failure rate across builds decreases, starting at 10% in release 36 and finishing at 6% in release 50.

Most of the build activity are tests, representing 95% of the build computation time.

We can see that globally, 5% of the build time goes to build compilation builds whereas 95% goes to testing, with the latter type of build being the more time-consuming.

The build activity — the number of builds, the failure rate, and the build computation time — depends on the branch.

Table 3.4 shows that there are more test than compilation builds. We also observe that some branches triggered more builds than others: for instance, the Try branch having the most buildernames in total, followed by Mozilla-inbound, and so on. Fail% represents the failure rate in those builds. We can see that the Try branch has the most failures, having 18% of build breakage for the compilation builds phase and 12% for the test builds. The other nine top branches show around 8% failures for compilation builds and 4.5% failures for test builds.

The four selected branches cover 50% of the builds and 70% of the total build computation time.

The Mozilla build team recommended us to study four major branches, as presented in Section 3.2.3. Before following that recommendation, we show here how representative those branches are regarding the overall activity.

We can see that three of those branches are in the top four of the build activity. This proves that those are important branches and that many builds are triggered from them. The Mozilla-release branch comes at the ninth place, still being relevant across the total 46 branches. It is also the last branch before integration, making it an interesting case to study, even though it has lesser builds than the three other studied branches. Table 3.3 shows that those four branches cover 70% of the build jobs. As seen in Table 3.4, the total build computation time of those four branches represents 70% of the total build computation time on the studied period. We can also observe that those branches have similar behaviour regarding build types and build failure as the other branches.

For this reason, we consider those four branches of interest as representative of the overall

activity on all branches. The following RQs focus on these four branches. We assume that this subset of branches will be representative of the global activity of all branches and that their activity corresponds to 70% of the overall activity across all branches.

RQ1.How does build activity evolve across time in Firefox?

Motivation:

In this section, we want to evaluate the global build automation activity over the four branches of interest in the Firefox project. We aim to get more understanding about how the overall build automation activity evolves over time as more tests and build configurations are added to the build system.

Approach:

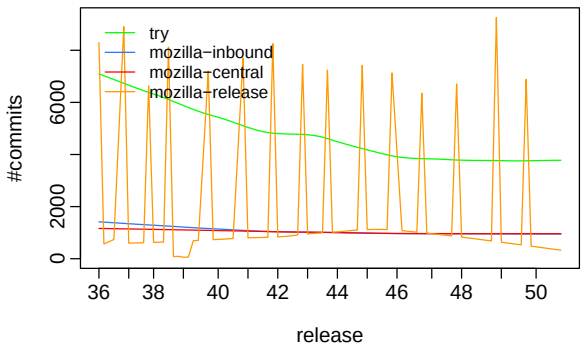
We analyze the activity within the Firefox project, counting the total number of pushes, commits, and builds per week. In each figure, we represent an approximation of the weekly activity using *loess*[3] (locally estimated scatterplot smoothing), in order to summarize the trend of global evolution. For the Mozilla-release branch, we observed several outliers related to the number of commits near the end of a release cycle. To deal with those outliers, we base the loess approximation on the non-outliers points. Then, we added peaks where the outliers appear.

Results:

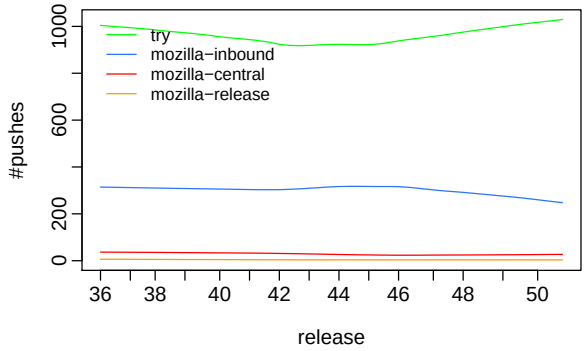
The number of builds per week increases significantly on the four branches over time, while the number of pushes and the number of commits per week decreases significantly over time.

Try has the most commits followed by Mozilla-inbound/Mozilla-central then Mozilla-release. Mozilla-release; the latter however peaks at the end of each release. Figure 3.2a shows that the Try branch's number of commits per week reduces significantly (alpha=0.01) over time with a small effect, starting at 7k during r36 until stabilizing at 5k in r45. The Mozilla-inbound and Mozilla-central decrease significantly over time with respectively a large and medium effect, and a median around 1k commits per week. The Mozilla-release usually shows lower numbers (500 to 1k commits a week), but has peaks near the end of each release (up to 8k commits per week), since the branch is used the most when releases are about to

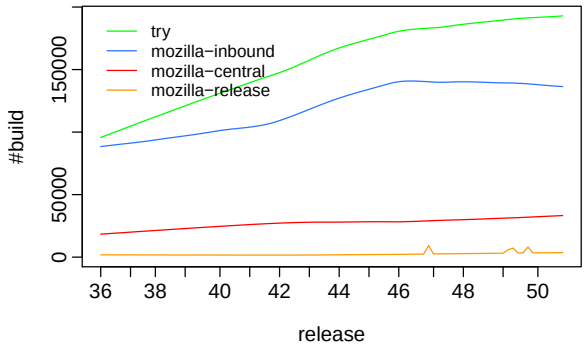
[3]The R loess function: `https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/loess` (accessed 2020-05-01)



(a) Number of commits by week



(b) Number of pushes by week



(c) Number of builds by week

Figure 3.2 Summary of Firefox's build activity by week of releases 36 to 50.

be shipped.

Despite the significant decrease in number of commits, the number of pushes per week is quite stable on the four branches, with Mozilla-central/Mozilla-release having the lowest number of pushes throughout. Figure 3.2b shows that the mean number of pushes by week on the Try branch is the highest, at around 1,000, followed by Mozilla-inbound at 250, then Mozilla-central around 35, and finally Mozilla-release under 10. the latter branch does not show peaks as it did in Figure 3.2a.

Although each branch has different trends regarding the number of builds per week, they are all increasing significantly (alpha=0.01) over time. The further the branch is in the integration path, the lower the number of builds per week. As shown in Figure 3.2c, the Try branch has the highest number of builds a week, starting at 100k and increasing significantly to 200k with a large effect. Then comes Mozilla-inbound, starting at 90k and increasing with a large effect to 140k. Mozilla-central increases significantly with a medium effect from 20k to 35k. Finally, Mozilla-release starts at 2k and finishes at 3.5k with a small significant

increase. While it is natural for build activity increase over time as a project grows, other factors like the number of contributors, supported configurations and environments, etc. also play an important role. Furthermore, old builds might not be relevant anymore and not highlight potential errors anymore at some point in the evolution. Efforts could be invested in detecting such builds and removing them to reduce the impact of the increase over time.

From these observations, we conclude that the number of builds per week increases significantly over time, even if the number of commits and builds are significantly decrease. Even though the commit activity decreases and less changes are submitted to the Firefox project, more and more build are triggered. That increase in build activity could be reduced by applying potential reduction techniques, related to test selection or build result prediction. We study in the next question the evolution of the activity per push and per commit to understand how the increasing build activity is distributed across the commits.

RQ2.How many builds are triggered per commit?

Motivation:

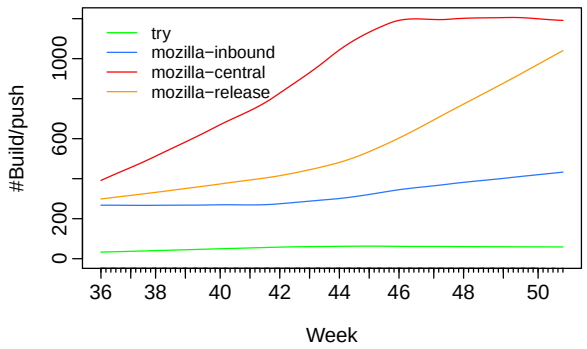
In *RQ1*, we evaluate the evolution of the build automation activity. We observed that the build activity increases over time, where the number of commits per week decreases. In this question, we evaluate the build inflation by commit over time. Since Mozilla triggers builds by push (sequence of commits) instead of by commit, the build activity is shared across the commits in the sequence.

Approach:

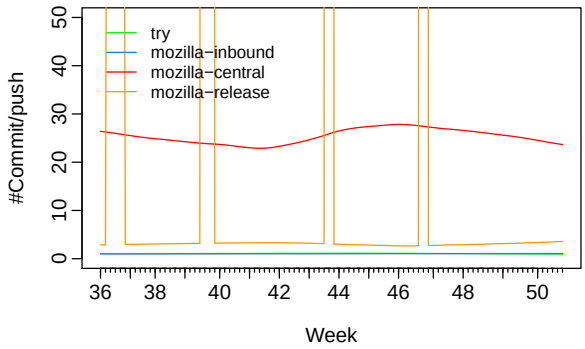
We introduce the concept of Effort Multiplier (EM). A commit's EM is the number of builds triggered by the automated build system for that commit. We need two other metrics to compute EM: the number of builds per push and the number of commits per push, referred to as #build/push and #commit/push, respectively, from now on. The build data has the information on the push related to each build job, and the commit data has the information about the push to which each commit belongs. From there, we compute the number of builds per commit (EM) as the #build/push divided by the #commit/push for the same push. The concept of EM's purpose is to evaluate the build activity generate by ech individual commit, since the activity should be split across all the commits in a push.

Results:

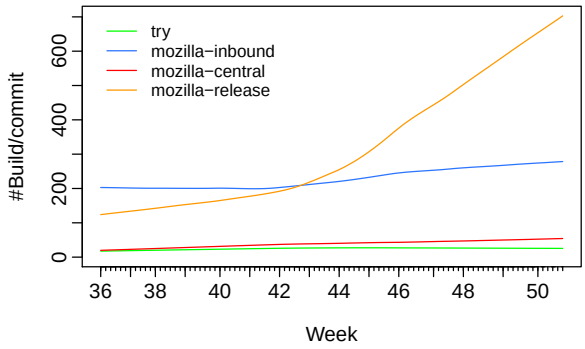
The #build/push of Mozilla-central stops increasing from release 45, while #build/-



(a) Number of builds by push



(b) Number of commits by push



(c) Number of builds by commit

Figure 3.3 Loess regression of the median values on each branch over weeks of releases 36 to 50.

push of Mozilla-release and Mozilla-inbound increases faster from that point on.

Mozilla-inbound branch's #build/push increases from 250 to 450 over time, as shown in Figure 3.3a. Mozilla-central increases from 400 in r36 to 1.2k in r45 then stabilizes. Mozilla-release branch's #build/push increases super-linearly from 300 to 1k over the studied period. From Figure 3.3a and Table 3.5, the #build/push's median on the Try branch starts around 30 in r36 and increases linearly to 60 in r50.

Mozilla-central's #commit/push oscillates between 20 and 30. The Mozilla-release branch's #commit/push is below 10, with major peaks mainly at the end of releases. It can be seen on Figure 3.3b and Table 3.5 that the Mozilla-central branch has a #commit/build's median varies between 23 and 28. The Mozilla-release branch has its #commit/build's median at 3, and show peaks up to 5,000 at the end of some releases. The height of those peaks is not shown on the figure for visibility. In contrast, the median #commit/push is at 1 on the Try and Mozilla-inbound branch, which means that commits on those branches typically are built individually, although the maximum number of commits

Table 3.5 Medians of the different measures by branch

Branch	build-push		commit-push		EM	
	r36	r50	r36	r50	r36	r50
Try	30	60	1	1	18	25
Mozilla-inbound	250	450	1	1	200	280
Mozilla-central	400	1,200	28	23	20	55
Mozilla-release	300	1,000	3	3	125	700

in a push can reach up to 10 for those branches.

The number of builds per commit (EM) increases significantly on the four branches, with a large effect. As can be seen in Figure 3.3c and Table 3.5, the Try branch's EM increases from 18 to 25, the Mozilla-inbound's EM branch increases linearly from 200 to 280 and Mozilla-central from 20 to 55. Mozilla-release's EM increases super-linearly from 125 to 700.

Since RQ1 learned us that the number of builds per week increases over time, while the number of commits and pushes do not, we interpret the EM's results as showing how each change triggers more and more builds over time, increasing with a large effect over time. As a project increases over time, the number of builds triggered naturally increases. However, old builds often remain in the build automation systems, regularly triggered even though they do not bring information about potential faults anymore. There is a build inflation over time and the build automation system should consider this increase to manage the available resources to run the builds, as well as the time and money consumption. Furthermore, in RQ0, we identify that 95% of the build activity is composed of testing build: for instance by applying test selection and prioritization techniques to build automation, less builds might be triggered and build maintainers could reduce the build activity. In the next question, we want to estimate the cost of that build automation and what part of that cost is actually valuable regarding error detection and correction.

In Appendix A, the detailed results for each metric (#build/push, #commit/push, EM) can be found. The results are split, having one figure by branch. Each week's results are represented with a boxplot of the results. Appendix A.1 shows the #build/push results, appendix A.2 shows the #commit/push results and appendix A.3 the EM.

RQ3.How much does a single commit in Firefox cost?

Motivation:

Although many researchers recognized the value of builds as a cornerstone of continuous integration activity [10] [11], little attention has been paid to their cost [18, 19, 21]. In this research question, we intend to estimate the cost of individual commits in a given branch. From there, we can evaluate the impact of each developer's changes on the build automation and give an evaluation of the cost brought by each change. The cost is computed based on the computation time of build automation multiplied by a cost factor depending on the build type and platform. We chose to estimate a cost to have a physical representation of resource consumption that speaks to developers and team leads.

Approach:

For this section, two approaches are used to evaluate the cost. First, a cost based on AWS on-demand 2020's pricing[4][5], as John O'Duinn did in his post in 2013 [6], which provides an upper-bound price estimation. Second, a cost based on the Microsoft Azure pricing from 2020[6], which provides a lower-bound estimation.

The build cost of each commit is calculated as its total build computation time multiplied by a cost factor. This factor varies depending on the platform the build is run on (Linux, MacOS, or Windows) and the type of build (compilation or test). The computation of the total build time is counted in machine hours as if all the builds happened consecutively. In reality, it is worth noticing that the build team at Mozilla uses a quite complex infrastructure[7] of build based on parallel jobs in more than 250 machines in order to reduce the time required to build new packages.

Table 3.6 Pricing [USD] of build computation time per hour, depending on the platform and the type of build (compilation or test).

Platform	AWS On-demand		Azure	
	Compilation [USD/h]	Test [USD/h]	Compilation [USD/h]	Test [USD/h]
Linux	0.60	0.096	0.0508	0.0508
MacOS	0.60	0.096	0.0508	0.0508
Windows	1.08	0.118	0.1428	0.1428

AWS Approach: The price of the Linux platform comes from the Amazon AWS 2020's pricing. Based on John O'Duinn's blog post [6], we selected the on-demand prices, which are the prices

[4]For compilations (general1.medium): `https://aws.amazon.com/fr/codebuild/pricing/`
[5]For tests (m5.large): `https://aws.amazon.com/fr/ec2/pricing/on-demand/`
[6]Microsoft Azure Pricing calculator: `https://azure.microsoft.com/en-ca/pricing/calculator/`
[7]see treeherder interface at `https://treeherder.mozilla.org/`

for renting the virtual machines on the spot when needed: the prices are indicated in Table 3.6 (price in USD per hour). O'Duinn's post [6] estimated the cost of MacOS and Windows in-house builds to zero since Mozilla already owns the machines. However, in-house runs need to consider the machine cost, the energy consumption, and the salary of the maintainers. These in-house costs are difficult to estimate since they are not public knowledge and depend on many factors. We thus decided to get AWS's costs for the MacOS and Windows platforms as well since the use of AWS should be similar or lower to the use of in-house machines.

Azure Approach: Using Windows Azure calculator, we estimated the cost of renting virtual machines with a 3-year-reservation (which gives a discount of ∼57% in comparison to the "Pay as you go" option). Windows platform builds' cost is 0.14 USD an hour and other platforms' cost (Linux and MacOS) is 0.05 USD an hour, as shown in Table 3.6.

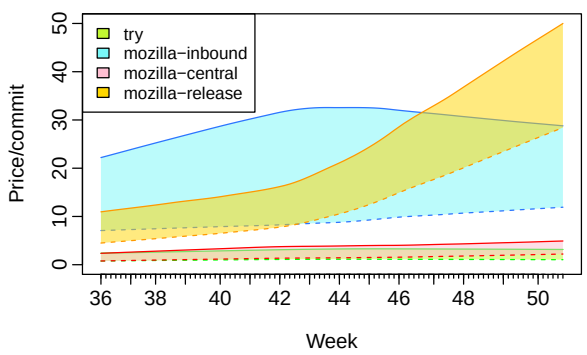
The fail-fix cost is the cost of failed builds and fixing builds. Furthermore, we want to estimate the part of the cost that was really useful to developers, either by flagging a failure or by validating that a prior failure has been fixed [31]. We, thus, introduce the concept of fail-fix cost as the cost of failed builds and fixing builds (i.e., first succeeded build after a failed build of the same name). One could argue that the fail-fix cost covers those builds that provided the largest return-on-investment of build infrastructure.

Figures in this section will present the results by branch for both AWS and Azure.

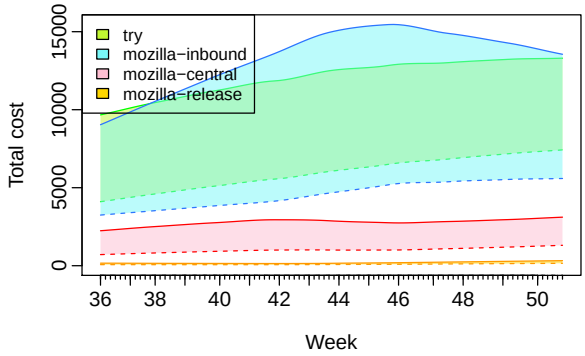
Results:

The Try and Mozilla-central branches all show median price between 1 and 5 USD per commit, while the median cost on the Mozilla-inbound branch increases significantly from 7-22 USD to 11-28 USD per commit and on the Mozilla-release branch from 4-11 USD during r36 to 28-49 USD. All branches show a significant increase with a large effect regarding cost of build automation per commit.

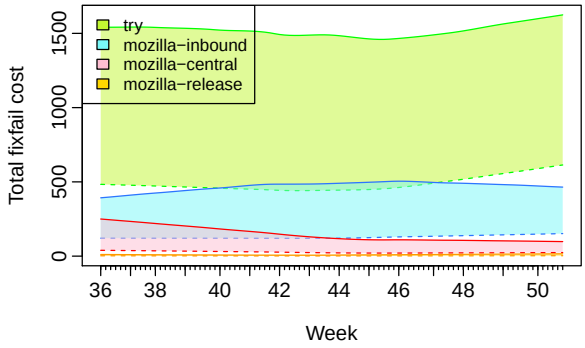
In Figure 3.4a, it can be seen that the cost of a single commit varies a lot from one branch to another. Try is cheaper than others, since the branch is a pre-review branch, where any developer can commit to test his changes: fewer builds are triggered so to keep that pre-review testing cheap. The Mozilla-central branch's price per commit is close to Try's because the branch is intermediate in the integration process: many builds have already past so more commits can be gathered into each push, reducing the over cost/commit. On the other hand, Mozilla-inbound is more expensive since the branch is at the beginning of the integration path, where many builds need to be tested to validate the code changes, and Mozilla-release is more expensive since the branch is the last step before integration, where the changes need



(a) Price (USD) by commit



(b) Total price (USD) by week



(c) Total price (USD) of failure and fix by week

Figure 3.4 Loess regression of the median values for (a) and total for (b) and (c) with lower-bound (Azure approach) and upper-bound (AWS On-Demand) on each branch over weeks of releases 36 to 50.

to be validated one last time.

The total cost per week on Mozilla-inbound and Try increases significantly with a moderate effect over time from 4k-10k USD to 6k-13 USD. Mozilla-central and Mozilla-release increase significantly with a moderate effect, staying bellow 3k USD and 250 USD respectively.

As seen in Figure 3.4b, on the Try branch, the total cost by week increases significantly (alpha=0.01) over time from 4k-8k USD to 8k-13k USD a week. The total cost of fail-fix by week stays under 15% of the total cost. The Mozilla-inbound branch's total cost by week increases significantly with a medium effect over time from 3k-9k USD to 6k-13k USD and fail-fix cost stays around 5%. The Mozilla-central branch's total cost by week varies between 700-2k USD and 1k-3k USD a week. The fail-fix cost slightly decreases over time from 40-250 USD (15%) to 20-100 USD (2%). The Mozilla-release increases significantly with a small effect from 70-160 USD to 150-300 USD a week. The fail-fix cost stays around 10 USD, lower

than 10% of the total cost.

The fail-fix total cost by week is a small part of the total cost, and becomes less significant as the branch goes closer to integration.

The closest a commit gets to integration, the lower the chances are that the commit still has build breakage.

In Appendix B, the detailed results for each metric with the AWS approach can be found, and Appendix C gathered the detailed results for the Azure approach. The results are split, having one figure by branch. Each week is results is represented with a boxplot of the results.

3.4 Discussion

While automated build systems are meant to reduce the time needed to integrate code changes, we observe that globally, the number of builds increases over time, and so does the cost of build automation. This is not due to an increase in the number of commits (in fact, we observe the opposite, i.e., a decrease in the number of commits), but instead, this comes from builds being added to the automation system over time to cover the evolution of the build necessities (i.e., different configurations of compilations/tests on different combinations of platform versions). Those increases have a large effect, showing that effort should be invested into reducing it.

Even if build automation increases naturally as a project grows, we observe that a large portion of builds are tests, making it possible to apply test selection and prioritization techniques to reduce the build activity. Furthermore, less than 15% of build cost is related to valuable jobs, i.e., jobs bringing information about the introduction or resolution of build faults, again showing that build reduction could be integrated into the build automation system. Developers should realize the impact their changes have on the build automation cost and build managers should continue to work on test selection/prioritization and make sure they do not run out of resources to run the builds.

3.4.1 Build cost per commit

Similar to many organizations, Mozilla has used the push mechanism to reduce its build activity. However, our numbers show that despite this mechanism, the number of builds still increases linearly; this suggests that other measures might be required to keep the build cost under control.

The Mozilla release pipeline is based on a hierarchical branch organization, which allows

having derivative build automation regarding each branch depending on its relevance as well as its importance in the integration process.

However, even with those adjustments, there is still work to be done on the build automation. Even if the Effort Multiplier is reduced by gathering commits into pushes, we found in RQ2 that the #build/push still increases over time, showing that this approach might not be enough, leading to more resources being allocated to build automation over time.

The cost of builds per commit increases as the number of builds increases, recalling CI is not cost-free. Through our study, we identified that pre-review branches and mid-integration branches cost less by commit, where the start of integration and the end of integration branches cost more. However, regarding the total cost, the further into integration the branch is, the less the branch cost per week, and the less failure-fixes happen. In fact, commits that got further into integration are less likely to break builds and have already pasted the filters of previous branches.

3.4.2 Total cost of builds per week

We evaluate the total cost per week of the Mozilla Firefox project on our branches of interest and observe that it increases over time. Considering the total build activity in RQ0, the total cost of all branches and all projects can only be larger than the cost obtained for the four branches in RQ3. Since the studied branches cover 70% of the total computation time and assuming the other branches have similar behaviour as the four studied branches (regarding build activity and platform usage), the cost we compute in RQ0 is a lower bound of 70% of the total cost. As such, the estimated weekly cost of a commit on the Firefox project overall branches increases from 11k-27k USD to 21k-42k USD per week.

3.4.3 Fail-fix cost

Furthermore, our analysis of fail-fix cost shows that only less than 10% of the builds are actually bringing information to developers. There is a large margin of improvement that is possible here on the other 90%, and thus we could improve the fail-fix cost rate on projects. This provides additional support for various lines of research, some already underway. For instance, developing strategies to identify which commits can be CI skipped in certain circumstances [32]. We identified that about 95% of builds are test builds, approaches such as test selection and prediction could be adopted to perform build prioritization and selective builds [12]. This could help the build system to reduce the build effort without removing the interesting fail-fix builds.

On the other hand, our estimation of the fix-fail might be impacted by noise such as unreliable builds, where the build failed not because of the code change but because of the build itself. If the broken build was rerun, the results would change and the build would be successful. This is what we call a brown build: those build are unreliable and should be corrected and removed from the CI process. Detecting those would reduce the build activity and thus the cost. This is what we study in next chapter.

3.5 Threats to Validity

Mozilla has a hierarchical branch system that creates a path to integration. Since we study four specific branches, which were identified as more interesting by members of Mozilla build team, the other branches were excluded from our paper. Furthermore, our approach may not be generalizable to projects without a branch hierarchy, and, thus use just a main branch, since any commit would ultimately be pushed to the main branch. Using other build infrastructure that Buildbot, or other cloud environment would also give other results.

The number of commits by pushes is counted from Mozilla's mercurial repositories. Mozilla may have performed some refactoring of their commits (i.e. rebasing or squashing), which would result in fewer commits in the push. Also, about 2% of the pushes do not match any commit on the mercurial repositories. The #commit/push might be slightly lower than in reality.

Regarding the type of results, we considered any result other than success to be a failure. While warnings are inherently not failures, however, they still bring information to developers about the need to do a correction to avoid that warning. As such, our analysis considers warnings to be failures, which might artificially increase the valuable part of the cost. Warnings represent 2% of the all dataset, whereas true failures represent 3%.

Concerning the cost of a commit, we use two cost approximation approaches (AWS on-demand and Azure) with pricing from 2020. If we had the cost of the period studied as well as updates by day or hour, the results would be more accurate. Builds on the other platforms (MacOS and Windows) are run in-house. Having an accurate estimation of the in-house cost based on energy consumption, the machine cost, and lifetime, as well as the maintainers' salary would be more accurate.

The fail-fix cost is estimated as the total cost of failed builds and fixes (first success following failing builds for a given buildername). Our definition of fail-fix might be mistaken if some failure is actively ignored or if a fix is someone that hid the error to correct it later. Some tests might also be flaky, and, thus, showing unreliable results.

3.6 Conclusion

The findings from this chapter validate our thesis hypothesis: we observe that the build activity and cost of the Mozilla Firefox project increases over time and should not be neglected. In fact, over the course of the 20 months we studied, the total cost of build activity increases from 11k-27k USD to 21k-42k USD per week. Furthermore, we show that less than 15% of that investment is actually valuable since it shows either a failure (thus, a potential error in the code change), either a fix from a previous failure.

Much work can be done to improve the build automation system and reduce the cost of CI. Since, about 95% of the build activity is test builds, many techniques as test selection and prioritization could be applied.

In the next chapter, we propose an approach to reduce the build activity and improve the CI system by detecting brown builds.

CHAPTER 4 BROWN BUILDS, A BUILD REDUCTION SOLUTION

In this chapter, we present a build activity reduction solution to improve the CI system. This validates the hypothesis that there should be effort invested into reducing the build activity and cost by at least applying this method. We present a model to detect brown builds, which are unreliable builds, and we evaluate it on six closed-source projects at Ubisoft that are known to contain brown builds. Such detected brown builds can be ignored or corrected by the CI maintainers, making the CI system more reliable and efficient, since less false negative results would block the integration process.

4.1 Introduction

In Chapter 3, we identify the fail-fix cost of automation build, which is the cost of failed builds and fixing builds (first successful builds after a failure for a buildername). One would assume that this cost is irreducible since those builds are needed to detect errors.

However, build results are not always reliable: some builds fail due to factors other than code changes, such as network, disk access, flaky tests, etc. Such builds with unreliable results are known as **brown builds** (or flaky builds).

A clear signal of brown builds within a CI is when the CI results change between runs without any code or configuration changes. In other words, the same process is run several times on the same code but does not yield the same results (pass/fail) every time. A preponderance of brown builds can lead developers to lose trust in the CI system: when developers face a failed build, instead of correcting their change, they trigger manual reruns of the builds, where the build might be a genuine failure and not a brown build. However, just relying on rerunning each build multiple times is not realistic, given the finding in chapter 3, where we showed that builds do not come for free. Reruns use resources and are a waste of productive time, as the developer must wait for the reruns to finish.

The fail-fix cost brings value to an organization, except for brown builds. Our definition of fail-fix from chapter 3 includes brown builds in the fail-fix cost. The value is thus artificially increased by brown builds and removing this type of unreliable builds will reduce fail-cost of the CI and reducing the overall cost.

In this chapter, we present a build activity reduction proposition, focusing on the case of brown builds, named Brown-Detector. We first study the ratio of brown builds on several projects to confirm the need for such an approach. The brown ratio can give us an idea

of the remaining value brought back by CI based on Chapter 3. Then, we will study the performance we can achieve with our approach.

This project was supported by La Forge, Ubisoft's research department in Montréal. The analysis is based on in-house closed-source projects with different development and CI activities. The need of a brown build detector was already identified by developers prior to the beginning of the project. They already lost trust in the CI. In fact, developers are more likely to rerun a failed build before checking if an error actually was introduced in the code by the code changed. Identifying those brown builds before sending the failed results to the developer might increase the trust in the CI system and improve the integration time.

Our approach leverages the logs produced within the CI. We extract vocabulary from the jobs' logs and reduce as much as possible the noise in that vocabulary. Words are reduced to their root and some patterns (like URL, paths, SHA, etc.) are replaced by identifiers. In addition, words are reduced to their root using stemming [33, 34]. The resulting vocabulary is then used to represent each CI job. The words in each CI jobs are transformed into a vector-based representation using the TF-IDF algorithm [35], which is an algorithm based on the frequency of a word in a document compared to its frequency in all the data-set. Then, boosted tree-based classifier [36] is trained on the data, and we can predict if a job is brown or not.

We validate the prototype on a large number of configurations of our model's hyper-parameters, comparing the performance to three baselines.

4.2 Brown-Detector Algorithm

In this study, we assess brownness of build jobs from the log files generated as the build's trace. Our approach is language-agnostic as it relies only on log files. The creation of the model is fully-automated to avoid potential errors that could occur during manual steps. However, it should be noticed that even if the approach is language-agnostic, the model is not and its performance depends on the log file content: some languages might use log files less efficiently, showing unrepresentative messages, which makes the brown build detection less effective.

To do so, we split our methodology into several steps, explained in the following sub-sections. The first step (section 4.2.1) is feature extraction from the log files, extracting words, and series of normalized words. Then, the vectorization step (section 4.2.2) aims to select the most relevant features from step 1, representing each log file with a uniform set of features. The following step (section 4.2.3) is the creation of the classifier model.

4.2.1 Feature extraction

Once the log files are gathered, we automatically extract the vocabulary of each file according to the following rules. Each replacement rule is described with a regular expression (Regex) using Python's format[1]. An example of a log file and of application of those rules can be found in Table D.1 (in Annexe D), where changes are marked in bold characters.

Rule1 All URL and file-paths are replaced by a known string. This way, we could still know a URL or file path was at a given position in the log file but disregarding what the URL or path was, which reduce the dimensionality of the vocabulary. We use the following regexes to identify URLs and file-path in log files.

(Regex url: `https?://[\S]+`)

(Regex path: `[\S]+[\.  / \\][\S]+`)

Rule2 Commit IDs (identified as a series of characters containing at least one letter and one number) are replaced by a known string. As for H1, we could still identify the presence of such a word, without having the impact of variation in those strings.

(Regex commit: `[\d\w]*\d\w[\d\w]*` and `[\d\w]*\w\d[\d\w]*`)

Rule3 Once the first two rules are applied to the file, we replace any non-letter characters with spaces, to separate all words. This removes any remaining special characters that do not contribute positively to the vocabulary.

(Regex non-char: `[_\W]+`)

Rule4 We add space before groups of uppercase characters to separate words in camelcase. For instance, the word "SeriesOfWords" becomes "Series Of Words". We are only adding spaces in front of groups of uppercase because we do not want to separate uppercase words: i.e. "WORD" will not change since the word is a group of uppercase.

(Regex group-upper: `[A-Z]+`)

Rule5 We remove English stop words [33] to reduce the impact of too common words of vocabulary.

Rule6 Finally, we apply a stemming algorithm [34] to extract the root of each word, reducing again the vocabulary's dimensionality due to plural or conjugated words.

[1]Python regex format: `https://docs.python.org/3/library/re.html`

On average, these rules reduce the size of the log file by 50% for the six projects we studied.

Once all those rules are applied to the data-set, we separate the text into words and perform an n-gram extraction (with $N = 1$ and $N = 2$). Using n-gram allows us to catch series of words that have meaning rather than focusing on single word only [37]. As an example, given the phrase "This is an example", the uni-gram extraction yields the list ["this", "is", "a", "example"] and the bi-gram extraction yields the list ["This is", "is an", "an example"].

Next, we computed the word count of each word and series of words by file. At this point, each build job is represented as a dictionary of features (words or series of words) by the number of occurrences in the trace. The dictionaries vary in size depending on the variety of the words in the log and the size of the log.

4.2.2 Vectorization

In this step, we create a uniform representation of the data, where each file is represented by a relevant set of features. The features will be either words, series of words, or other metrics. Those other metrics are contextual metrics about the job, regarding the previous runs of the given build job/commit (number of reruns prior to the current job, number of failure and success among those reruns) and regarding the prior jobs/commits (number of jobs/commits since the last brown job was observed).

The data-set is split into three sub-groups: a training set, a test set, and a validation set. This follows classical practices in machine learning.

The vectorization is generated on the training set, then applied to the other two sets. It means that if a word (or series of words) is present in the testing or validating sets but not on the training set it won't be part of the created vectors. In addition, we avoid information contamination by gathering all the jobs a of a commit in the same set: if we did not, the performance computation might have been too optimistic, since builds of a same commit might have impacted the prediction, which is not representative of a real utilization of the model.

TF-IDF computation

We used TF-IDF to represent each feature instead of the word-count to reduce the impact of huge files on the numbers [35]. This algorithm needs the frequency of each word in all the data-set.

The TF-IDF algorithm generates for each feature f in each file's directory d the value as

follows:

$$TFIDF_{fd} = TF_{fd} \cdot IDFf \tag{4.1}$$

The frequency by document (TF) is computed by selecting in each file/document's directory, d, only the features, f, appearing in the training set. TF is thus the rate of occurrences of a feature, f, in a directory, d, divided by the total of words in the directory, d.

$$TF_{fd} = \frac{d[f]}{\sum_{t_i \ \in \ d} d[t_i]} \tag{4.2}$$

The inverse frequency of each word in the corpus (IDF) is computed as follows:

$$IDF_f = log \frac{|T|}{|\{d_i \ : \ f \ \in \ d_i\}|} \tag{4.3}$$

where, $|T|$, is the size of the training set (number of build jobs) and $\{d_i \ : \ f \ \in \ d_i\}$ the number of file directories in the training set where directory i, d_i, contains the feature, f. The IDF computation does not have any information about the validation and test sets. It will thus not contaminate the results and performance of our algorithm.

Table E.1 in Annexe E shows an example of TF-IDF applied to a corpus of three logs.

KBest feature selection

Because the total vocabulary size of the files is similar (95%) to the number of files by project (a mean of 41k words for ngram 1 and 2, with a mean of 43k files) , we used feature selection to reduce the size of the vocabulary of our data. We used SelectKBest feature selection from the sklearn package [38] in Python.

The algorithm uses a score function and the number of features to be selected. The score function we used is the default function, which is the ANOVA F-value computation [39] between the feature and the classification.

The number of features to select was chosen experimentally as described in section 4.4.

Other algorithms were considered for feature selection, for instance, infogain [40] and correlation computations [41]. Both algorithms have similar results as KBest regarding the selected features in the case of the studied projects (other approaches would choose the same features, with about 5% of other features, while getting similar performance) but they are computationally taxing. Because the selected features are similar and infogain and correlation are

slower to compute we only report the results using SelectKBest.

Iterative computation

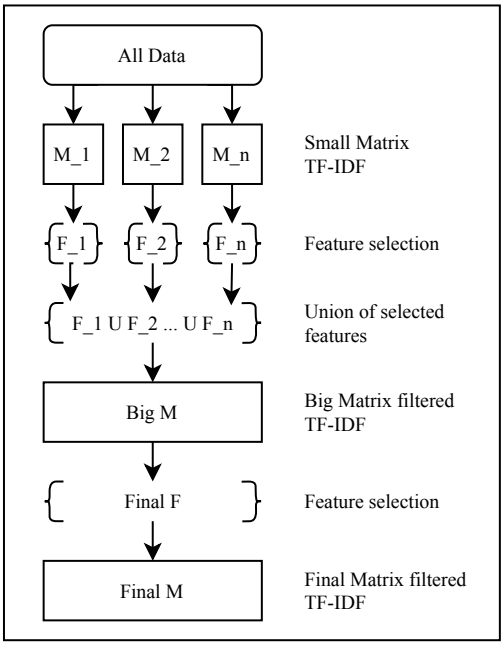


Figure 4.1 Iterative vectorization algorithm

Another problem we face is the resources used to compute the model. The features gathered from the job's traces create a vocabulary that increases super-linearly as we add new traces and thus new vocabulary. Generating a matrix jobs/features with all the vocabulary. Then applying KBest on it would need large memory and computation resources. Also, once KBest is applied, most of that data will be removed and the memory will be released.

To limit the memory complexity, we applied an iterative computation of the vectorization, as summarized in the Figure 4.1. This approach will be explained in this section. We do the feature selection on the training set only, so their is no contamination between the model creation and the validation and test set.

First of all, we split the data into sub-matrices of a selected number of builds from the training set, so that each build is at least in one matrix. The matrix values are computed by the TF-IDF algorithm. The size of the sub-matrices was set to 1000: this value was chosen based on the studied projects, to have at least 10 observations of brown builds by set for all projects.

From there, the KBest algorithm is applied to each sub-matrix to extract sets of selected

features F_i. Then, the sets of features are united in a single set F_{uni} (see. (4.4)).

$$F_{uni} = \bigcup_{i \in subMat} F_i \tag{4.4}$$

Then, a matrix is generated with the selected features F_{uni}, again using TF-IDF.

The feature selection algorithm is finally applied one last time to clean the union set F_{uni} from features that should not have been chosen in the total matrix, and thus deal with interactions between the sub-matrices. The resulting set is named the final feature selection F_{final} and a final matrix is generated with TF-IDF. The same F_{final} is used on the validation and test set to build their respective vectorization.

The iterative approach is used to reduce the computation time of the vectorization step. For this approach to be acceptable and thus similar to just applying once the TF-IDF computation to the full matrix, we need to validate hypothesis **Vec-H1**. In fact, if the size of the united set F_{uni} (sets of bests features selected on all sub-matrices) is similar to K, the selected features must be similar among all the matrices. This hypothesis was confirmed when experimenting on the performance and the size of the united set F_{uni} did not exceed K by more than $(50 \pm 5)\%$, which we considered as neglectable.

Vec-H1 If the united feature selection F_{uni}'s size similar to the size of the F_i sets (K, since all sets composed of the K best features), the iterative approach can be accepted (cfr. (4.5) and (4.6)).

$$|F_{uni}| \simeq |F_i| \tag{4.5}$$

$$\simeq K \tag{4.6}$$

Other metrics

In addition to vocabulary features, we also compute the number of reruns, failures, and successes before each build job. To do that, we gather each job by commit ID and job name, then ordered them by creation date. Each job then got a triplet of the values, that could be added to the previously computed vectorization. Another metric we added is the number of commits since the last brown job.

4.2.3 Classification

The classification is done on a training and a validation set. The performance is estimated on a test set, which is composed exclusively of failed build jobs. In fact, the detection tool would be used only on failed jobs to assure that they are true failures or if they are brown.

Hyper-parameters that drive the learning process are used in our study and we report their impacts in section 4.4. One of those hyper-parameter is a filter that we apply on the sub-sets (training and validation sets). These filters have diverse impacts on the performance of our approach.

- *AllData*: No Filter, we take all the jobs.
- *TFails*: Filter on training set to take only failed jobs, validation still.
- *OnlyFails*: Filter on all sets to take only failed jobs.

Once a matrix is generated for the three sets (training, validation, and test set) using the KBest selection and TF-IDF computation, we used the XGBoost algorithm [36] (eXtreme Gradient Boosting) to predict the brownness of each build job. This algorithm is a supervised classification algorithm based on random forest. This algorithm has the advantage to boost its parameters automatically to increase its performance based on a validation set. We did not explore other classification techniques, since random forest is well-known to have significantly good performances regarding supervised classification [42]. Further studies could be done on applying other classsification algorithms .

To boost the model, it takes a training set and a validation set. Until now, the validation set was not used in the creation of the model, since the vectorization algorithm is only based on the training data. This was done to have a representative validation set, which did not contaminate the matrix construction process. The model is thus not contaminated by the validation data and using this set in the XGBoost validation is representative relatively to the test set.

The classification is divided into a two-step model training, as can be seen in Fig 4.2. The first step includes only the features of the vocabulary. The second step takes as input the shap values of the first model and other metrics. The shap values [43] are a numerical estimation of the impact of each feature on a log file's prediction. A positive value will pushes the prediction towards brown, a negative value towards a real failure. We filter out the features that have a standard variation of shap value of zero.

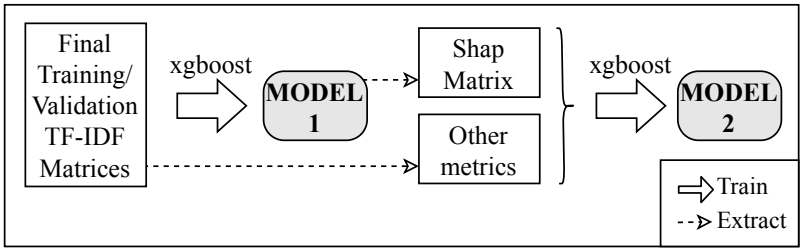


Figure 4.2 Two-step model training

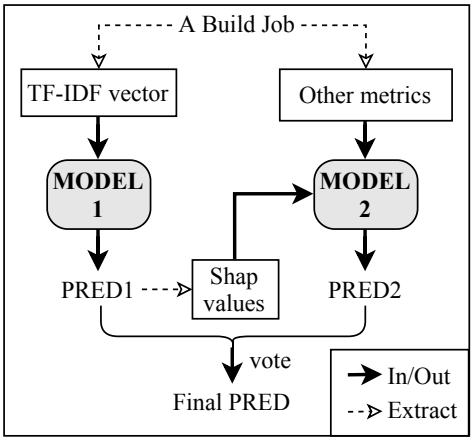


Figure 4.3 Prediction with two models

The separation in two steps allows us to evaluate the impact of the other metrics. In fact, for each build job in the test set, two predictions are computed, one for each model. This is shown in Figure 4.3. Each prediction is a real value between 0 and 1. If the prediction is close to 1, then the classification tends to be brown, where a if it is close to 0, it is safer. A threshold is tested to decide which prediction to chose, as will be discussed in the next paragraph.

Then, we apply a vote between both predictions to have a final classification of the build. The vote is split between the two models, with Pred_2 having a weight of α and Pred_1 a weight of $(1 - \alpha)$, as can be seen in (4.7). Giving more weight to one or the other prediction will give more or less impact on the other metrics. Then, the classification is done by choosing a threshold β at which the classification is chosen to be brown (see. (4.8)).

$$\text{Pred}_F = \frac{\alpha}{100}\text{Pred}_1 + \frac{100-\alpha}{100}\text{Pred}_2 \tag{4.7}$$

$$\text{Classification} = \begin{cases} Brown, & \text{if Pred}_F > \beta \\ Safe, & \text{otherwise} \end{cases} \tag{4.8}$$

Both hyper-parameters α and β are chosen experimentally during the validation phase and will be discussed in section 4.4.

4.3 Case study setup

4.3.1 Research Questions

We establish the following research questions for the brown build project:

RQ4 *What is the brown ratio in projects at Ubisoft?*

We want to see what the brown rate is on the studied projects to see what the impact of a brown build detector could be. We identify, in Chapter 3, that less than 10% of the total cost of Mozilla Firefox builds are attributed to fail-fix builds. This high cost of builds could be impacted by the presence of (unreliable) brown builds that should have been ignored or corrected . Thus in this RQ, we aim to identify the ratio of brown builds and the impact of their removal on the reduction of the build activity (and thus, costs) of selected Ubisoft projects.

RQ5 *Can we accurately detect brown builds in a language/project agnostic way?*

Although approaches have been proposed for flaky test detection, based on code dependencies [29, 44], dynamic code analysis [45], or test-smells, these are difficult to implement on the level of builds due to several languages and tools used. Furthermore, since flaky tests are not the only problem (a brown build could be due to a hardware problem, or a versioning problem of the build, etc.), we focused on trying to identify brown builds with a language-independent model. The model would be based on identifying the brownness of a build from the log file.

4.3.2 Data Sources

We gather data from several projects that are multi-language, have different purposes and log characteristics (size, vocabulary, construction). The point is to have projects as different

Table 4.1 Information on the projects studied.

Project	DevOps Platform	#month	#jobs	#failjobs
A	Gitlab	18	23k	3,008
B		29	63k	8,100
C		17	22k	5,986
D	TeamCity	3	88k	8,711
E		2	53k	1,310
F		22	9k	1,705

as possible so that the result can be generalized to any project. Six projects are considered through this chapter, as can be seen in Table 4.1.

In what follows, we present briefly each project. Since the projects are closed-source, information about the project size could not be disclosed.

Project A: This project is three (3) years old and has been supported by a team of three programmers. It consists of two-dozen micro-services developed in Go. The services are mainly focused on analyzing code, issue triaging, reviewers' recommendations and other activities related to the analysis of software production artifacts.

Project B: This project is five years old and supported by a team of five programmers. It is a desktop application that displays profiling information extracted from video-games. It is developed in C# and WPF (Windows Presentation Foundation). Because of the size of the profiling information to be displayed, the complexity of the application lies in optimizing the graphical threads.

Project C: This project is ten years old and has received contributions from more than 50 different developers in the last year alone. It is actively being developed and maintained by a team of 4 developers. This project is developed in C++ and provides low-level, cross-platforms (PC, Consoles) and cross-engine mathematical functions to compute, for example, collisions.

Project D: This project allows the cross-platform, cross-engine creation of HUD (heads-up-display) in video games. It has been developed for the past 7 years by a team ranging from 2 to 8 developers. It is mainly developed in C++ but also has C# components (20% of the code base).

Project E: This project is a cross-platform, cross-engine pathfinding library that video games developers can leverage when they need their characters to automatically go from point A to point B. It uses classical path finding algorithms such as A* with video-game specificities

(e.g. not all paths are equal, a vehicle could be too wide to fit in an alleyway and the path would be ignored for instance). This project is maintained by a team of 6 programmers and is developed in C++.

Project F: This project is an asset (e.g 3D textures, animation, sound, ...) web search engine where the artists can search for past assets in order to reuse them. It has a backend separated in a dozen C# micro-services and a frontend developed using a javascript framework. It is supported by a team of 20 developers. These developers are also in charge of other software projects that are not part of our studies. Consequently, less than 100% of their time is invested in this particular project.

4.3.3 Scraping

We extract data about build jobs from two DevOps platforms: Gitlab and TeamCity (three projects by platform, as can be seen in Table 4.1). Both of those platforms offer large API documentation.

The data that is gathered from both platforms for each build job is shown in Table 4.2.

Table 4.2 Scrapped data for each build job on each project.

Information	**Type**
job's ID	Integer or hexadecimal
job's name	String
job's status	success or fail
job's start time	date
code change's ID	Integer or hexadecimal

GitLab

For each project of interest, we get the corresponding project ID on GitLab. From there, we gather information on the build jobs via this API call:

`api/v4/projects/{projectID}/`**`jobs`**

We filter out the jobs with a status different from success or failure (for instance, running and canceled). We also remove jobs with a null start time, which means they are not started yet at the scraping time.

Then, we scrape the job traces for each job ID through the following call:

`api/v4/projects/{projectID}/`**`jobs`**`/{jobID}/trace`

TeamCity

With TeamCity, the project ID is the name of the projects we are searching for directly. Similar to the Gitlab process, we first gather some information about the build jobs, using the next API call:

```
/app/rest/builds?locator=project:{projectID}
```

To select the needed information, we add to this API call the parameter "field", which retrieves for each job the same information as for GitLab.

Again we filter jobs only to get jobs with status success or fail and removed unfinished jobs.

The trace is retrieved from this call:

```
/httpAuth/downloadBuildLog.html?buildId={jobId}
```

4.3.4 Brown build identification

To create our model, we needed to have a posteriori classification of the log files. In fact, the point of our brown detector is to identify brownness and true failures before they are manually rerun. We based the identification on the definition of brown build: a brown build job is a job that changed results without a change to the source code (i.e. after a rerun) [46].

Thus, we grouped jobs by name and commit ID. Then, if at least one job name changes its result for the same commit ID, then all the runs for that job name and commit ID are marked as brown. This means the flaky builds are identified by historical reruns. All the other jobs are marked as safe, even though some brown builds may remain from jobs not rerun enough to show a brown behaviour. This means that some failures incorrectly might be considered as a true failure instead if a brown build.

4.3.5 Validation approach

Hyper-parameter tweaking

From the methodology, we identifiy a list of hyper-parameters to tweak. Those are gathered in the following list, with a small recap of their purposes and the range of values that were chosen.

- $F(set)$: filters to apply on the sets.
  (Range: *AllData* (no filter), *TFails* (only fails in the training set) and *OnlyFails* (only fails in all sets))

- N: Number Ngram to consider.
  (Range: [1], [2], [1, 2])
- K: Number of features to be chosen by the feature selector.
  (Range: 100 to 300, by 25)
- α: Percentage of predictions repartition between the two models for the classification.
  (Range: 0 to 100, by 10)
- β: Threshold for the classification.
  (Range: 10 to 90, by 10)

For all the studied projects, we train models with each hyper-parameter configuration. We use cross-validation to validate the results with the data-set, as will be explained in the following section.

Cross-validation settings

The purpose of this validation is to identify the best hyper-parameters for our model and evaluate if the classifier's performance is relevant.

To do the cross-validation, all build jobs are given a group label in a range G. We chose $G = 10$, splitting the data into 10 groups given a group number from 0 to 9, with the same number of brown builds in each group. Each group is composed of randomly selected jobs across the data-set, respecting the commit ID separation (see **Iso-H1**): no jobs from the same commit are separated into different sets (training, validation, and test sets).

Iso-H1 Commit ID isolation: Two build jobs on the same commitID cannot be separated into different data-set groups.

Also, the ratio of brown jobs of each group is similar to the global ratio in the whole data-set.

From there, the **CV** approach is applied:

CV In this approach, the three needed sets (training, validation and test) are defined as follows (for each iteration i):

- Train set: all groups but group i (90% of data-set);
- Valid set: half of group i (5% of data-set);

- Test set: the other half of group i (5% of data-set).

Each set of hyper-parameters combinations is used to train the vectorization step. The vectorization is computed ten times excluding each time one of the groups as can be seen in the Algorithm 1. Then, the excluded group is split into two sub-groups to get a validation set and a test set. The training set and validation set are used for the classification, since XGBoost uses a validation set to improve the performance of the model trained on the training set, and a prediction is done on the test set using the produced model. The classification is done two times so that both halves of the sub-group are once used as a validation set, then test set.

Algorithm 1 Cross-validation – Appr1

```
group = assignGroups(data)
for i in range(G) do
    TRAIN = data[group != i]
    vectorization(TRAIN)
    other = data[group == i].split(2)
    for j in range(2) do
        VALID = other[j]
        TEST = other[1-j]
        classification(VALID,TEST)
    end for
end for
```

Through this technique, for each set of hyper-parameter, $2 \cdot G$ models are generated and all jobs are predicted once by one model. Also, only G vectorizations are computed, which is the longest step.

The approach **CV** takes a week to compute for a machine with 128Gb of RAM and 4.0 GHz of processor, on the six studied projects with 91 configurations of hyper-parameters.

Performance metrics and model comparison

To compare models with each other, we used the F1-score. The hyper-parameter set with the highest value is considered to be the best set.

The F1-score's equation is given at (4.9), usually used in the literature [47–49]. It is computed as the harmonic mean of the precision and recall.

$$F1 = \frac{2}{precision^{-1} + recall^{-1}} \tag{4.9}$$

$$precision = \frac{TP}{TP + FP} \tag{4.10}$$

$$recall = \frac{TP}{TP + FN} \tag{4.11}$$

$$LocOp_p = \arg\max_c F1(c) \tag{4.12}$$

$$GlobOp = \arg\min_c \sum_{p \in proj} (LocOp_p - F1(c))^2 \tag{4.13}$$

We computed local and global optimization of the hyper-parameters based on the F1-score.

For each project p, the local optimization *LocOp* is the hyper-parameter-combination c that optimizes p's F1-score of the prediction, cfr. (4.12). The optimal combination set may be different for each project.

The global optimization is the hyper-parameter combination c that minimizes for all projects the sum of the squared difference between the F1-score of the combination and the local optima, see (4.13). This gives the overall best configuration while minimizing the different with the local optimization by project.

Table 4.3 Baselines. (With BFR the brown rate among the failed build jobs by project)

Baseline name	**Description**	**Precision**	**Recall**	**F1-score**
Random50	Predict with a 50% chance the build job to be brown.	BFR	$\frac{1}{2}$	$\frac{1}{\frac{1}{BFR}+2} = \frac{BFR}{1+2BFR}$
RandomB	Predict with a BFR% chance the build job to be brown.	BFR	BFR	$\frac{1}{\frac{1}{BFR}+\frac{1}{BFR}} = \frac{BFR}{2}$
AlwaysBrown	Always predict the build job to be brown.	BFR	1	$\frac{1}{\frac{1}{BFR}+1} = \frac{BFR}{1+BFR}$

The local and global optimizations are then compared with the baselines, described in the Table 4.3. Those baselines are intuitive predictions, based on random predictions. To our knowledge, no other project did brown build prediction in a language-independent way. Since the projects we studied are multi-language in nature, it was not possible to compare to the state-of-the-art baseline approaches, since those are language-dependent. We considered using flaky test detectors, for instance Deflaker~ [29] which covers Java code. However, due to the many different languages used in the projects studied and the fact that Java is almost never used in the studied projects, it is not possible to put in place. Also, the Deflaker

approach detected flaky tests, so brown compilation (i.e. not test) jobs might not be covered detected.

4.4 Results

RQ4.What is the brown ratio in projects at Ubisoft?

Table 4.4 Metrics on the studied projects.

Project	#jobs	#failjobs	Fail ratio	Brown ratio	Brown ratio fails (BFR)
A	24k	3k	12.5%	7.2%	30.1%
B	63k	8k	12.7%	15.5%	37.5%
C	22k	6k	27.3%	2.4%	5.2%
D	88k	9k	10.2%	1.1%	5.6%
E	53k	1.3k	2.5%	5.1%	58.4%
F	9k	1.7k	18.9%	3.7%	10.1%

The brown ratio of all failing build jobs varies from 5% for proejct C and D to 58% for project E. This justifies the needs for an efficient brown build detector.

Table 4.4 shows metrics about the projects that we studied. The number of jobs (#jobs) was already referred to in Table 4.1 where we observed that the number of jobs varies from one project to another. The number of failed jobs (#failjobs) also varies from one project to another, and the fail ratio varies between 2.5% for project E and 27% for project C.

When looking at the brown build activity, we observed that the brown ratio ($\frac{\#brown}{\#jobs}$) varies between 1.1% for project D and 15.5% for project B, with a median at 5.1% for project E. Even if the percentage seems low for some projects, it is still representative of the overall build activity since each project has several thousands of builds.

Finally, we looked at the brown ratio among the failed jobs ($\frac{\#brown}{\#failjobs}$), referred to as BFR, which varies between 5% for project C and 58% for project E. The BFR value varies among the studied projects and we observe that a lot of failed jobs are actually brown, especially for projects A, B, and E.

RQ5.Can we accurately detect brown builds in a language/project agnostic way?

Our models are about two to four times better than the baseline *OnlyBrown* in terms of F1-score (95% confidence interval: 3.28 ± 1.09).

In Figure 4.4, we gather all the F1-score results for each hyper-parameter-configuration by project. We observe that the performance vary depending on the project. This is due to the

Table 4.5 Model performance.

Proj	Brown-Detector								**Baseline**											
	Local opti [%]				**Global opti** [%]				**Random50** [%]				**RandomB** [%]				**AlwaysBrown** [%]			
	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1
A	74	55	84	56	74	55	75	52	50	30	50	19	30	30	30	15	30	30	100	23
B	76	65	78	59	69	56	82	55	50	37	50	21	37	37	37	19	37	37	100	27
C	94	40	35	29	92	32	40	28	50	5	50	5	5	5	5	3	5	5	100	5
D	92	32	38	25	90	21	27	17	50	6	50	5	6	6	6	3	6	6	100	5
E	85	84	93	79	79	75	95	73	50	58	50	27	58	58	58	29	58	58	100	37
F	90	51	53	44	85	36	62	40	50	10	50	8	10	10	10	5	10	10	100	9
Mean	85	55	64	49	82	46	64	44	50	24	50	14	24	24	24	12	24	24	100	18

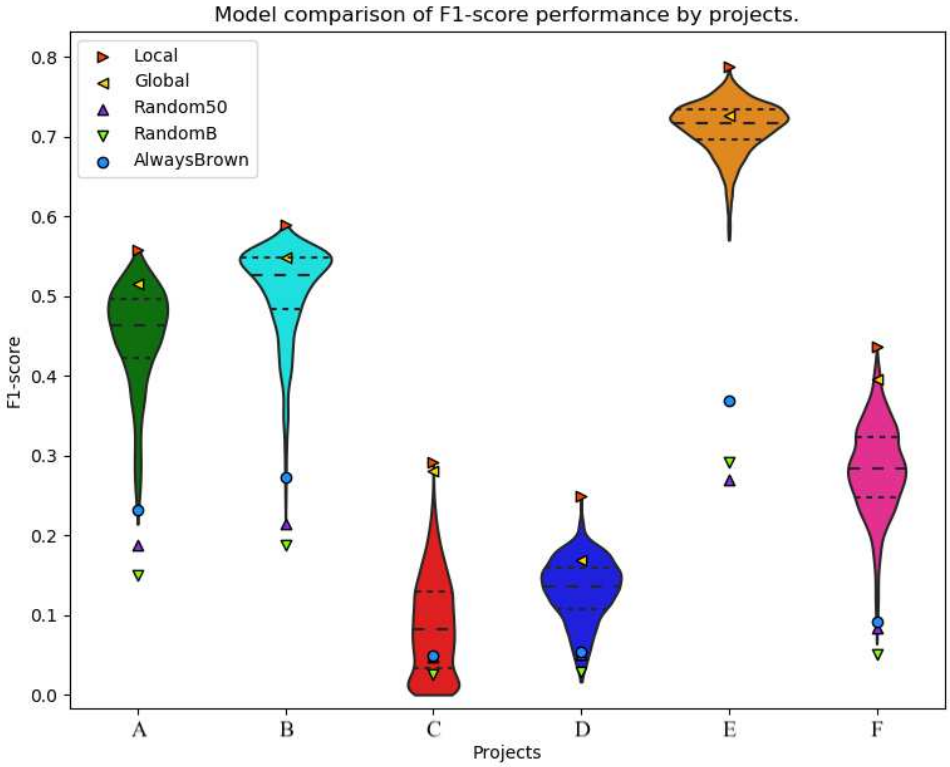


Figure 4.4 Optimization of the hyper-parameters.

unbalance of brownness in build jobs differing from one project to another. The higher the ratio of brownness in the failed build job is, the better the performance of our algorithm is. If the brown ratio is small (for instance, project c, d and f), there will be more false positives, and brown jobs will be more difficult to find, which makes the F1-score drop. However, the accuracy, shown in Table~4.5, stays over 69% for all projects, with a mean of 83%, which is about 30% higher than the Random50 baseline. Also, the performance varies a lot from one hyper-parameter-configuration to another in the same project.

The local and global best combination are indicated per project on the figure. The F1-score decreases a little (about 5%) when choosing a global set of hyper-parameters. However, the new score for all projects stays at least higher than the median, and higher than the third quartile for projects A, C, D, and F.

The reference baselines are also shown for each project. The F1-scores of those are always lower than the first quartile of our algorithm configurations, except for project C where they get a little higher.

Table 4.6 Hyper-parameters chosen through the optimization of F1-score.

Optimization		Hyper-parameters				
Type	Project	F	N	K	α	β
Local	A	TFails	[1]	300	0%	10%
	B	Fails	[1]	275	40%	20%
	C	TFails	[2]	300	90%	20%
	D	Fails	[1, 2]	100	50%	10%
	E	Fails	[1]	300	50%	30%
	F	Fails	[1]	225	100%	10%
Global	All	TFails	2	300	70%	10%

The baseline that gets the best F1-score is *OnlyBrown*. Yet, the mean F1-score of the global optimization of our models is between two and three times better than the baseline.

The results of the F1-score and other metrics (Accuracy, Precision, and Recall) are gathered in Table 4.5. The local optimization per project has a mean accuracy of 85%, with a minimum at 74% for project A and a maximum at 94% for project C. When looking at the global optimization, we lose 3% of the mean accuracy, this time with a minimum at 69% for project B and a maximum at 92% for project C. The precision's results for local optimization varies between 32% for project D and 84% for project E, where the global optimization vary between 21% for project D and 75% for project E, dropping from a mean accuracy of 55% to 46%. Regarding the recall, the local optimization gives us a recall between 35% for project C and 93% for project E, and the global optimization a recall between 27% for project D and 95% for project E. The mean recall stays at 64% in both optimizations.

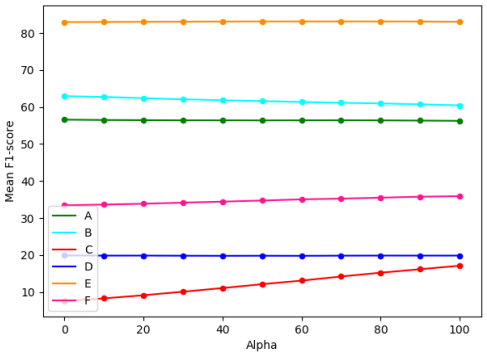
The best hyper-parameter-configuration regarding local and global optimization are shown in Table 4.6.

The hyper-parameters variation show similar impacts on the models' performance for all projects

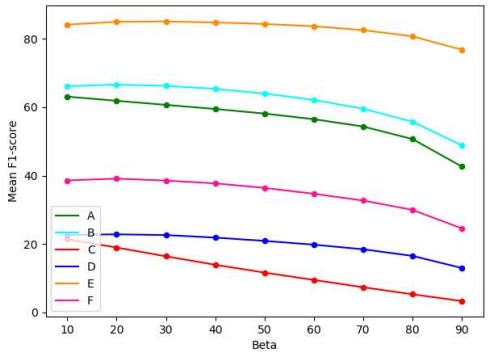
In Figure 4.5, we compare the impact of each hyper-parameter on the mean F1-score of our prototype by project. Each color corresponds to one of the six projects studied, corresponding to colors in Figure 4.4.

The Figure 4.5a shows the impact of α on the prototype performance. For most of the projects is small ($< 5\%$) , having the F1-score staying stable as α goes up until 80%, then it decreases about 5%. However, project C shows a totally different behaviour, where the larger α is, the better the F1-score gets. The global optimization's α value is 70%.

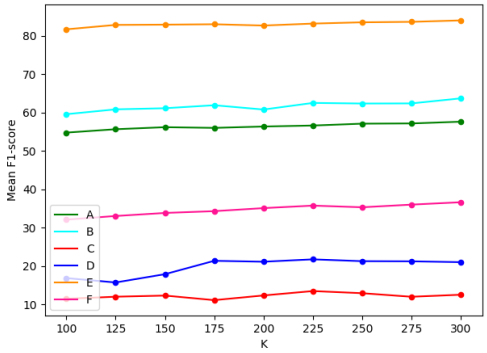
Next, Figure 4.5b shows the impact of the threshold β selection. This shows that the lower



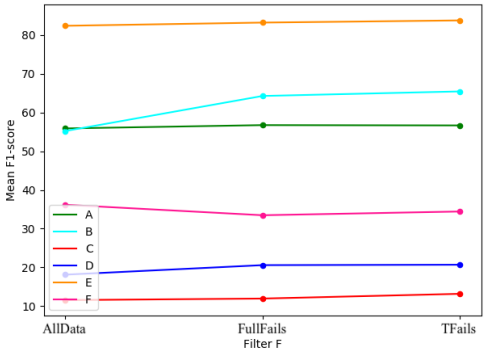
(a) α Weight of model$_2$ on the prediction



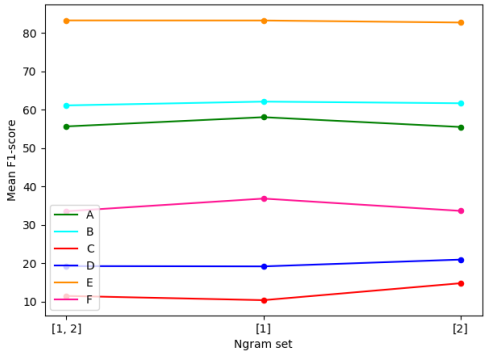
(b) β Threshold for the classification



(c) K Number of features selected



(d) $F(set)$ Filters on the sets



(e) N Number Ngram

Figure 4.5 Comparison of the mean F1-score performance by hyper-parameters.

the threshold is, the better the results are, having the best value between 10 and 30%.

The K parameter in Figure 4.5c shows that the more features are selected the better the results. However, the impact on the results is not that big and it costs a lot in terms of computation time, since the more features are taken into account, the longer the training of

the model gets.

Regarding the filters $F(set)$, Figure 4.5d shows that $TFails$ is slightly better for most projects. The difference is not that big, but the computation time for $TFails$ is much smaller than $AllData$ since the training set is smaller.

The n-gram N best value depends on the project. For most of the projects, $N = [1]$ worked the best.

4.5 Discussion

We observe that between 5% and 58% of failed build jobs are brown, depending on the project. This highlights the need to address this brown build problem and to propose a detection algorithm, at least in the studied projects. This forms a representative part of the build activity, especially on big projects with many contributors. Identifying and correcting those builds would improve the CI system and reduce the build activity.

Through our research, we suggest an approach that is language and project agnostic. This approach applies to any project using build automation and for which the build jobs' logs can be retrieved. In fact, the prediction is based on the log content and a lack of log information will harm the performance of the algorithm. Even if the approach is language and project agnostic, the model is not. Depending on the project's log usage, the results will vary. The algorithm is based on the word extraction of the jobs' logs, which allows the model to apply to any project, disregarding the project language, and build characteristics.

We manage to select hyper-parameters for the algorithm that locally and globally maximized the F1-score for the six studied projects. The mean F1-score for the six projects global optimization is 44%, which is more than two times the mean F1-score of the baseline *AlwaysBrown* (which always predicts failed jobs as brown). Regarding the accuracy of our model, global optimization gets a mean accuracy of 82%, which is about 30% better than the *Random50* baseline. The mean precision is at 46%, which is almost two times higher than all the baselines (26%) . The mean recall is 64% which is 14% higher than the *Random50* baseline. This performance prove that our model is reliable and helps detected brown builds efficiently.

By detecting brown builds, we can automatically trigger reruns before the developers get the first results of the build job. If the results changes, the job is then marked as brown and should be looked at by a CI maintainer. Until then, the integration system is improved by having the right result sent to the developers: the CI result is more likely to be right and the developers would rely more on the CI system and correct errors instead of manually

rerunning failed jobs and consume resources.

It could be used by CI systems on projects showing brown build behaviour. By training a model on past cases, the CI system would be more reliable and the build activity would be reduced. However, the algorithm would not work on projects not keeping jobs' logs or jobs not using jobs' logs.

4.6 Threats to Validity

Our algorithm is evaluated on six projects with various brown rates and build activities. We observe that the recall and precision decrease as the brown rate decreases. A project having a brown rate lower than 1% might not justify the need for our algorithm since the performance might not be sufficient.

The chosen validation approach does not do time isolation: a prediction on a build job might be based on jobs that happened later in the project lifetime. The performance might be more optimistic than if the time isolation was respected, and, thus, the training set always contains older jobs than the validation set and so on for the validation set with the test set.

Our brown job might be misidentified since we base the a posteriori identification on the reruns of jobs on the same commit-ID. Most of the build jobs are run only once for a commit ID, since brown builds are rare: those are systematically identified as real fails or successes. However, if a job is rerun at least once and changes results for the same commitID than the job is identified as brown. We might be missing some brown jobs from that, but all jobs identified as brown are sure to be brown. The performance can be impacted by that.

4.7 Conclusion

This chapter shows that their is a considerable amount of brown builds occuring on the six projects we study. Using our brown build detection model, we managed to identifty brown builds with a mean accuracy of 82%, a mean precision of 46% and a mean recall of 64%. By correcting those builds, the CI system would be more reliable, having less false negative build results send to the developers. The integration process becomes more efficient and quick, since brown builds would not block the integration path of the code changes.

CHAPTER 5 CONCLUSION

5.1 Summary of Works

With this thesis, we validate the hypothesis that build automation has a cost that increases significantly over time and that efforts should be invested into reducing that cost.

Chapter 3 studies the cost of CI over time through a case study on 20 months of Mozilla build data. The cost is based on the time duration of build automation, multiplied by a cost factor depending on the build's type and platform. We observe that the cost per commit increases over time, with Mozilla-inbound and Mozilla-central branches being the most affected. The total cost increases on all branches while the fix-fail total cost decreases over the studied period.

We find that even though measures are being taken to reduce the build load, for example by gathering multiple commits into pushes, as well as using branch hierarchy that iteratively built and tested commits, the build activity increases over time on all branches, as well as the EM.

The EM, which is the number of builds run per commit, increases over time. Together with the decreasing but high number of commits per week, this results in a non-negligible amount of money invested per week in builds and tests that cannot be ignored by team managers and developers. Furthermore, even less than 10% of builds spot failures: only 10% of the builds are valuable for the CI.

Those observations show that more proactive build management could be done to reduce the build consumption and to make better use of the machine, time, and money resources. Many approaches can be explored to reduce build activity, and, thus, the overall cost of the CI system. The results of Chapter 3 are submitted as a paper to ICSME 2020 and are currently in the review process.

In Chapter 4, we propose a reduction approach by identifying brown builds, which are builds with unreliable results. Those builds do not bring relevant information to the developer since a failure result might be due to a build problem instead of a potential error in the code changes that are built.

The detection algorithm that we present is language and project agnostic, and is tested/evaluated on six different projects, each of them using many different languages, types of files, and having different purposes. By optimizing the model's hyper-parameters for the set of

projects, we manage to get a mean accuracy of 82%, a mean precision of 46%, a mean recall of 64% and a mean F1-score of 44%. The approach is language and project agnostic, however the model is not, since the prediction is based on the log file content: if a project does not provide log files with sufficient information, this might lead to less efficient predictions, depending on the project setting and language used. The results are compared with several baselines, showing that the accuracy per project is about 30% higher than the *Random50* baseline, while the F1-score is more than two times higher than for the *AlwaysBrown* baseline. The precision is almost two times higher than the three baselines and the recall is 14% higher than the *Random50* baseline. Those performance numbers show that our algorithm detects brown builds efficiently.

When encountering a failed job, the model predicts the chances for it to be a brown build or a true failure. If it is predicted as brown, it can be automatically rerun. If, in fact, the result changes and the job succeeds, the developer does not lose time rerunning the job manually. Also, the build is marked as brown and can be corrected by the CI manager to improve the CI system. Eventually, when brown builds are corrected by CI managers, the CI systems become more reliable for the developers, who become more likely to trust the builds' result and not rerun builds himself.

As a conclusion, we validate our research hypothesis by studying the build automation activity and cost by commit at Mozilla, showing that the cost increases over time and is non-negligible. We also identify that less than 15% of that cost is valuable for the developer (fix-fail cost). Furthermore, we validate that the build activity could be reduced, for instance by detecting brown builds and correcting those builds them. We propose a detection algorithm and a globally optimized model over six projects at Ubisoft. Further studies could be explored on build reduction, to reduce the build activity even more, which would reduce significantly the cost of build automation in CI systems.

5.2 Future Research

In Chapter 3, we identify that the cost of build activity increases over time, and that there is a necessity to do build reduction. In Chapter 4, we approach one build activity reduction method, by identifying and correcting brown builds. Other approaches could be explored, regarding build prioritization or selection, to reduce the build effort while making the relevant build rate more significant. Since more than 95% of the build jobs are tests, approaches based on test selection and prioritization could be adapted to builds. However, builds bring a new dimension with environments being taken into account: build systems have to test several platforms and versions. The selection should thus also take into account the reliability and

pertinence of certain platforms and versions.

Regarding the brown build algorithm, more contextual metrics could be considered, regarding the state in a project's life where a commit is pushed on the system. Depending on that, the build activity might need to change. For instance, a build system at the end of a release should not behave the same way as during the release: the commit activity and the kind of changes done on a project differ over time and that could be quantified and use as a feature.

Furthermore, the time isolation of the training, validation, and test sets should be studied, to see how the performance behaves when adding that constraint. The time isolation constraint would be to verify that all jobs in the training set are older than in the validation set, and the jobs in the validation set are older than in the test set.

Testing our model on other data-sets with manually labeled brown builds would be interesting too, since it might show different results. However, this would need a specialist on a project as well as on the CI system. It would also need a lot of time to manually identify build jobs.

Finally, something we would like to study is the drifting concept, which is the aging of the data we use, the model, and the impact on the performance. Old data might be outdated and not bring any further information on the current state of the project and current prediction. Also, if a model is used for a certain period, when should it be retrained with more recent data? The impact of the drifting concept on our algorithm might highlight interesting parameters that still optimize the performance while improving the computation time and the model training, especially in the context of the video game industry where software projects can go on for several years and the developing behaviour shifts over time.

REFERENCES

[1] R. Ablett *et al.*, "Buildbot: Robotic monitoring of agile software development teams," in *Proceedings of the 16th International Symposium on Robot and Human Interactive Communication*, 2007, pp. 931–936.

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

[3] D. Ståhl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *Journal of Systems and Software*, vol. 87, pp. 48–59, 2014.

[4] M. Zolfagharinia, B. Adams, and Y.-G. Guéhéneuc, "Do not trust build results at face value: An empirical study of 30 million cpan builds," in *Proc. of the 14th Int'l Conf. on Mining Software Repositories*, ser. MSR '17, 2017, pp. 312–322.

[5] K. Gallaba *et al.*, "Noise and heterogeneity in historical build data: An empirical study of travis ci," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, 2018.

[6] J. O'Duinn, "The financial cost of a checkin (part 1)," Nov 2013, accessed in May 2020. [Online]. Available: https://oduinn.com/2013/11/20/

[7] B. Adams and S. McIntosh, "Modern release engineering in a nutshell – why researchers should care," in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 5. IEEE, 2016, pp. 78–90.

[8] M. Zolfagharinia, B. Adams, and Y.-G. Guéhénuc, "Do not trust build results at face value-an empirical study of 30 million cpan builds," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR).* IEEE, 2017, pp. 312–322.

[9] A. Memon *et al.*, "Taming google-scale continuous testing," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track.* IEEE Press, 2017, pp. 233–242.

[10] B. Vasilescu *et al.*, "Quality and productivity outcomes relating to continuous integration in github," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE'15, 2015, pp. 805–816.

[11] M. Hilton *et al.*, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proc. of the 31st Int'l Conf. on Automated Software Engineering*, ser. ASE 2016, 2016, pp. 426–437.

[12] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17, 2017, pp. 356–367.

[13] H. Seo *et al.*, "Programmers' build errors: A case study (at google)," in *Proc. of the 36th Int'l Conf. on Software Engineering (ICSE' 14)*, ser. ICSE 2014, 2014, pp. 724–734.

[14] L. Hochstein and L. Jiao, "The cost of the build tax in scientific software," in *Proc. of the 5th Int'l Symposium on Empirical Software Engineering and Measurement (ESEM' 11)*, 2011, pp. 384–387.

[15] A. Neitsch, M. W. Godfrey, and K. Wong, "Build system issues in multilanguage software," in *Proceedings of the International Conference on Software Maintenance (ICSM'12)*, 2012, pp. 140–149.

[16] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," in *Proc. of the 30th Int'l Conference on Software Maintenance and Evolution (ICSME'14)*, Sept 2014, pp. 41–51.

[17] I. Kwan, A. Schroter, and D. Damian, "Does socio-technical congruence have an effect on software build success?a study of coordination in a software project," *IEEE Trans. on Soft. Eng.*, vol. 37, no. 3, pp. 307–324, 2011.

[18] K. Herzig *et al.*, "The art of testing less without sacrificing quality," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, 2015, pp. 483–493.

[19] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 140–150. [Online]. Available: http://doi.acm.org/10.1145/1273463.1273483

[20] ——, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[21] S. Alspaugh *et al.*, "Efficient time-aware prioritization with knapsack solvers," in *Proceedings of the International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, ser. WEASELTech '07, 2007.

[22] T. A. Ghaleb, D. A. da Costa, and Y. Zou, "An empirical study of the long duration of continuous integration builds," *Empirical Software Engineering*, Mar 2019.

[23] Q. Li and B. Boehm, "Improving scenario testing process by adding value-based prioritization: an industrial case study," *ICSSP 2013: Proceedings of the 2013 International Conference on Software and System Process*, pp. 78–87, May 2003.

[24] M. Hilton *et al.*, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 426–437.

[25] K. Gallaba *et al.*, "Noise and heterogeneity in historical build data: an empirical study of travis ci," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 87–97.

[26] Q. Luo *et al.*, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 643–653.

[27] J. Bell *et al.*, "Deflaker: automatically detecting flaky tests," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 433–444.

[28] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: a study of java projects using continuous integration," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 821–830.

[29] J. Bell *et al.*, "Deflaker: Automatically detecting flaky tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 433–444.

[30] M. Beller, G. Gousios, and A. Zaidman, "Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *Proceedings of the 14th working conference on mining software repositories*, 2017.

[31] X. Jin and F. Servant, "A cost-efficient approach to building in continuous integration."

[32] R. Abdalkareem *et al.*, "Which commits can be ci skipped?" *IEEE Transactions on Software Engineering*, vol. 14, no. 8, pp. 1–17, 2019.

[33] List of english stop words. Accessed in February 2020. [Online]. Available: http://xpo6.com/list-of-english-stop-words/

[34] Stemmer golang package. Accessed in February 2020. [Online]. Available: https://github.com/caneroj1/stemmer

[35] J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242. Piscataway, NJ, 2003, pp. 133–142.

[36] Xgboost python package. Accessed in February 2020. [Online]. Available: https://xgboost.readthedocs.io/en/latest/python/index.html

[37] W. B. Cavnar, J. M. Trenkle *et al.*, "N-gram-based text categorization," in *Proceedings of SDAIR-94, 3rd annual symposium on document analysis and information retrieval*, vol. 161175. Citeseer, 1994.

[38] Sklearn selectkbest python package. Accessed in February 2020. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html#sklearn.feature_selection.SelectKBest

[39] Sklearn f-classif python package. Accessed in February 2020. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.f_classif.html#sklearn.feature_selection.f_classif

[40] R. Y. Chen *et al.*, "Ucb and infogain exploration via q-ensembles," *arXiv preprint arXiv:1706.01502*, vol. 9, 2017.

[41] J. Benesty *et al.*, "Pearson correlation coefficient," in *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.

[42] C. Tantithamthavorn *et al.*, "Automated parameter optimization of classification techniques for defect prediction models," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 321–332.

[43] Shap values documentation. Accessed in February 2020. [Online]. Available: https://shap.readthedocs.io/en/latest/

[44] W. Lam *et al.*, "idflakies: A framework for detecting and partially classifying flaky tests," in *2019 12th ieee conference on software testing, validation and verification (icst)*. IEEE, 2019, pp. 312–322.

[45] J. Ha *et al.*, "System to uncover root cause of non-deterministic (flaky) tests," Apr. 12 2016, uS Patent 9,311,220.

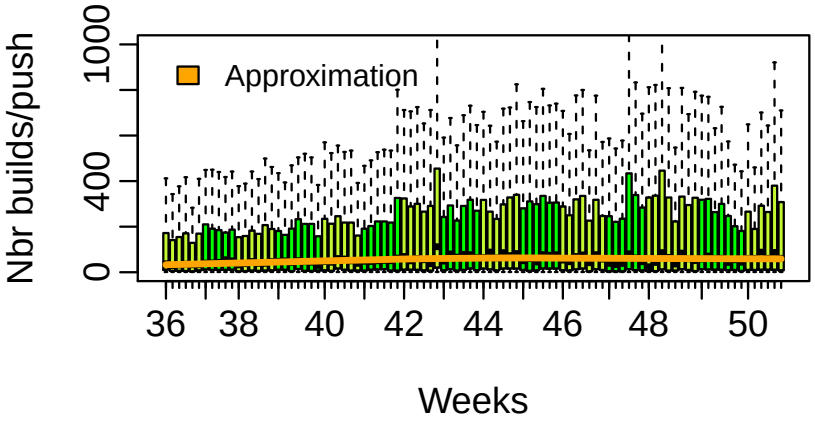
[46] T. Durieux *et al.*, "Empirical study of restarted and flaky builds on travis ci," *arXiv preprint arXiv:2003.11772*, 2020.

[47] A. Fujino, H. Isozaki, and J. Suzuki, "Multi-label text categorization with model combination based on f1-score maximization," in *Proceedings of the Third International Joint Conference on Natural Language Processing: Volume-II*, 2008.

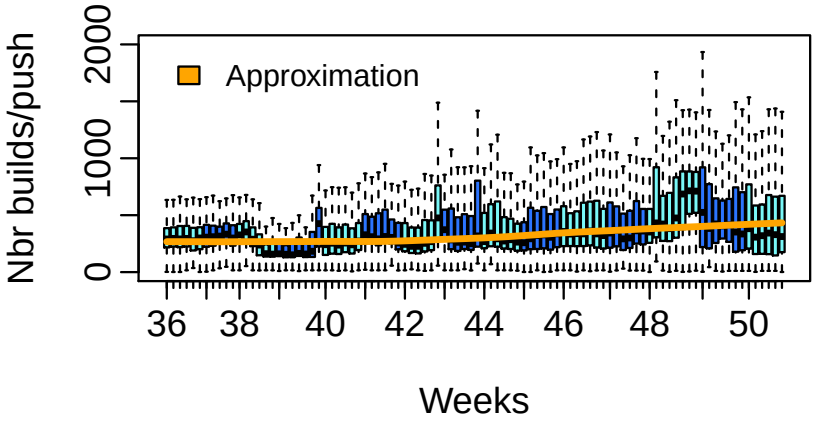
[48] H. Huang *et al.*, "Maximum f1-score discriminative training criterion for automatic mispronunciation detection," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 23, no. 4, pp. 787–797, 2015.

[49] Z. C. Lipton, C. Elkan, and B. Narayanaswamy, "Thresholding classifiers to maximize f1 score," *Machine Learning and Knowledge Discovery in Databases*, vol. 8725, pp. 225–239, 2014.

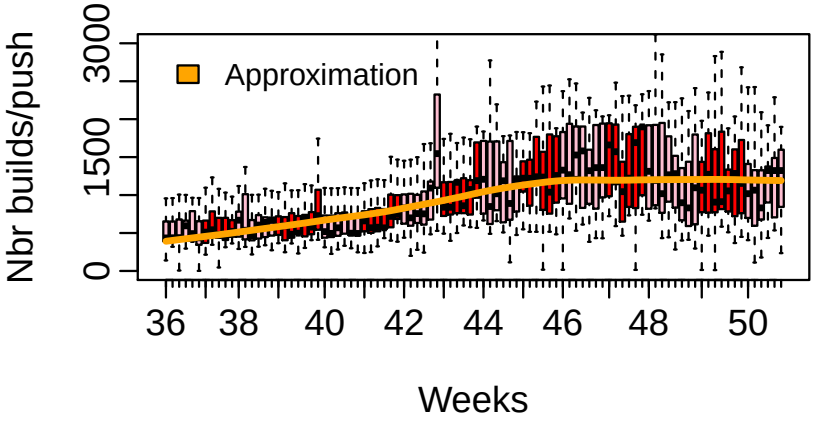
APPENDIX A COMPLEMENTARY RESULTS FOR RQ2



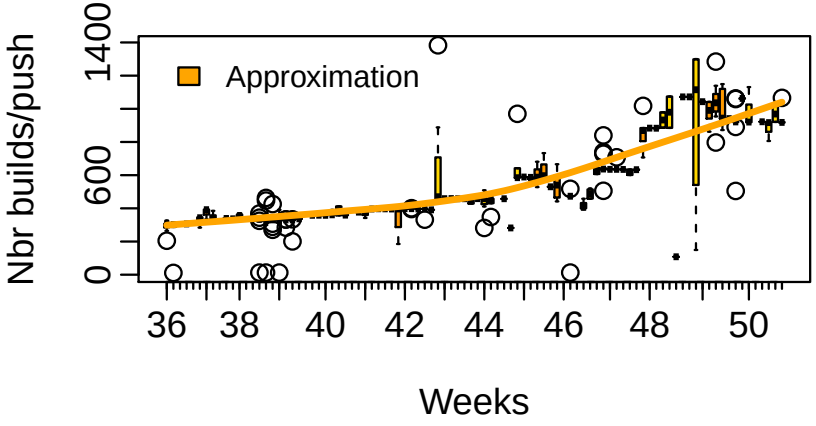
(a) Try branch



(b) Mozilla-inbound branch

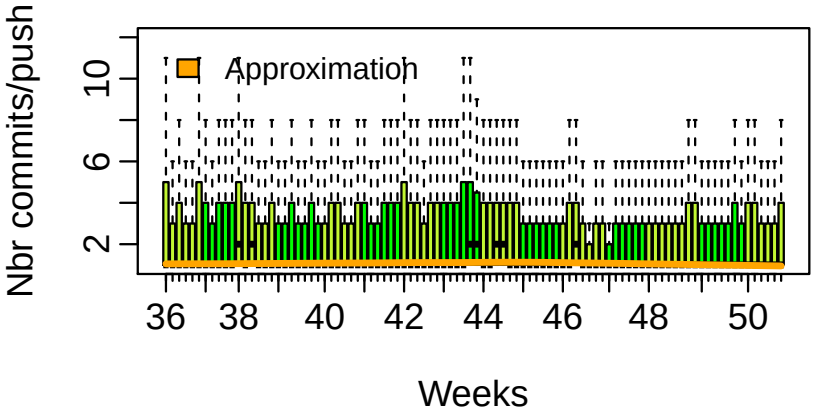


(c) Mozilla-central branch

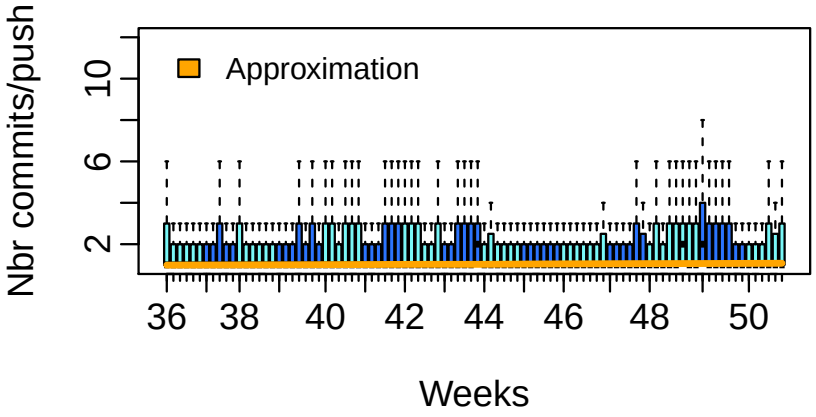


(d) Mozilla-release branch

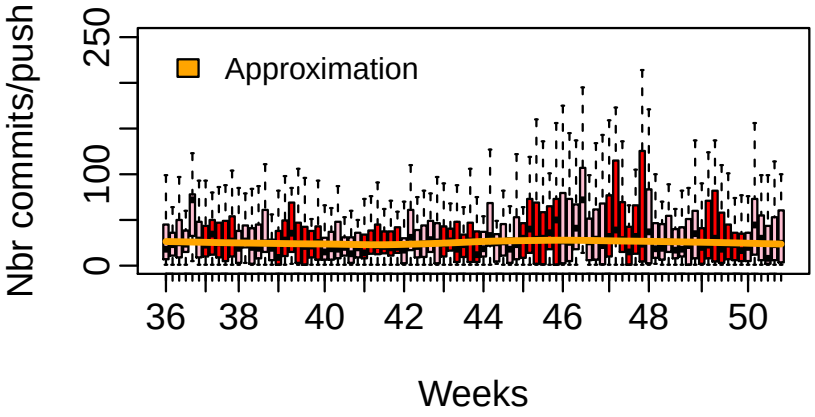
Figure A.1 Number of builds by push (#build/push) on each branch on each branch over weeks of releases 36 to 50.



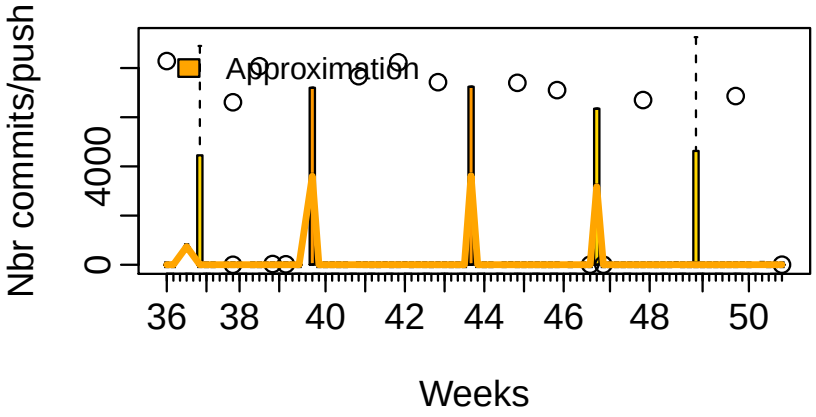
(a) Try branch



(b) Mozilla-inbound branch

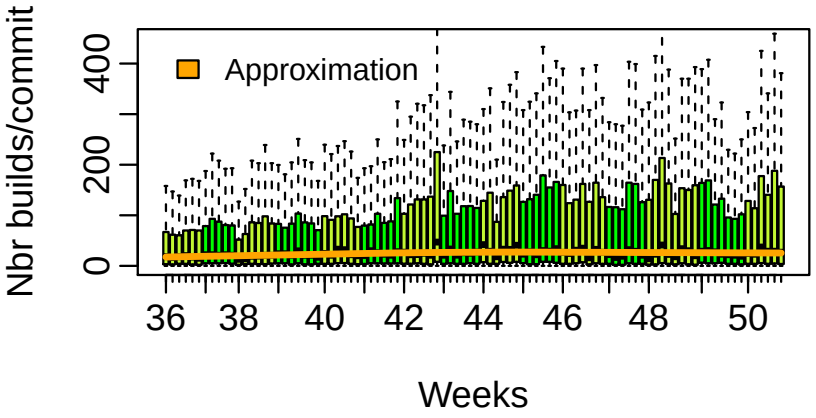


(c) Mozilla-central branch

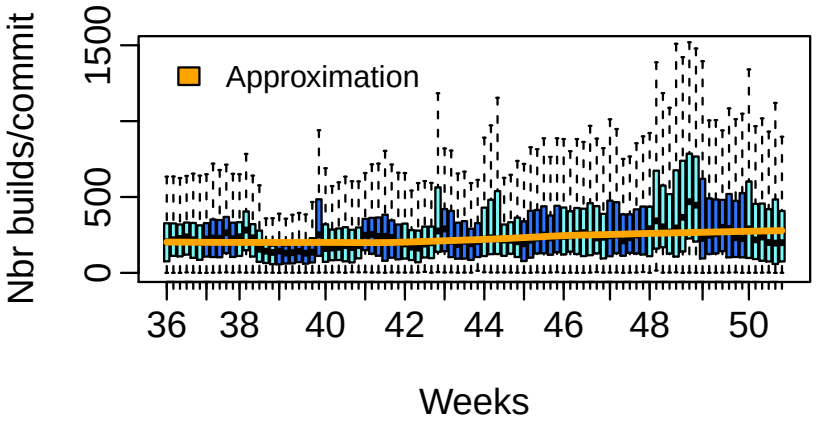


(d) Mozilla-release branch

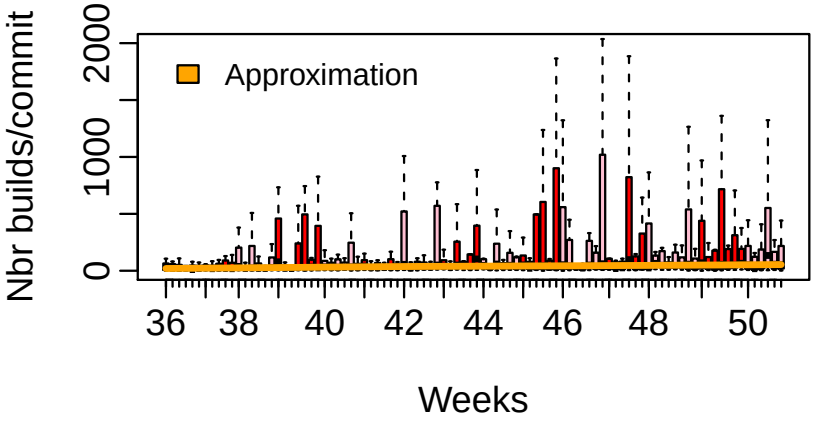
Figure A.2 Number of commits by push (#commit/push) on each branch on each branch over weeks of releases 36 to 50.



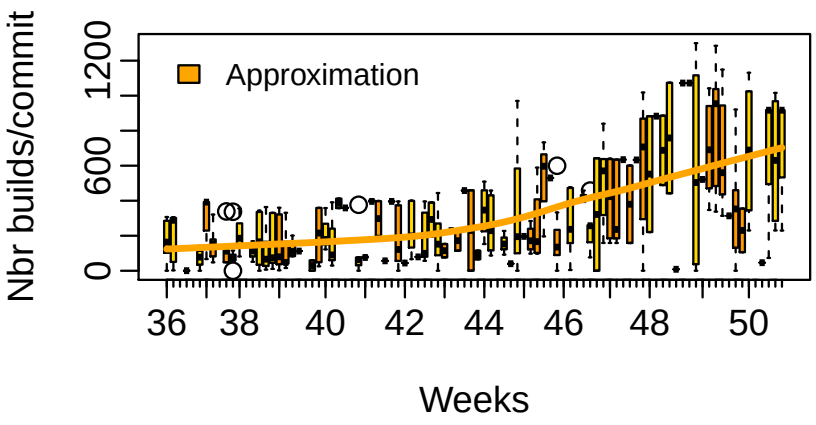
(a) Try branch



(b) Mozilla-inbound branch



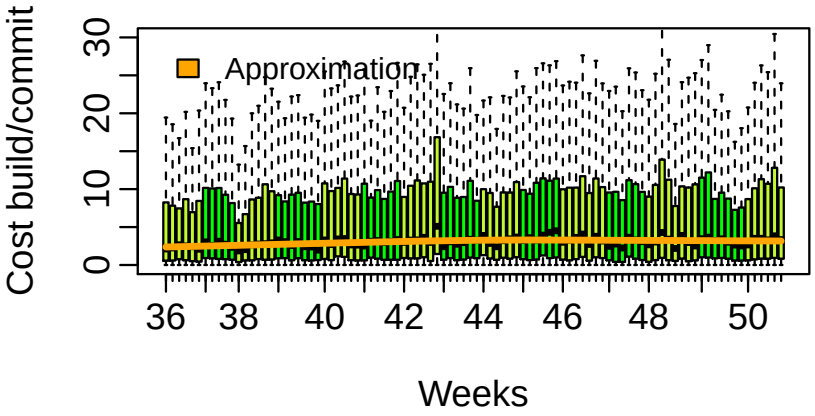
(c) Mozilla-central branch



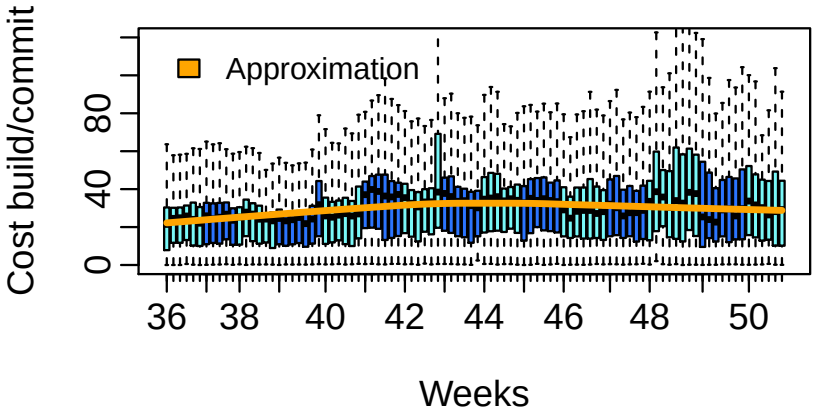
(d) Mozilla-release branch

Figure A.3 Number of builds by commit (EM) on each branch on each branch over weeks of releases 36 to 50.

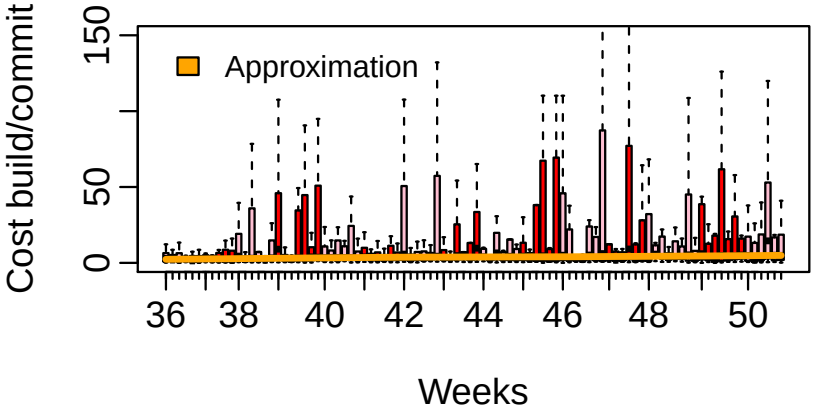
APPENDIX B COMPLEMENTARY RESULTS FOR RQ3, AWS APPROACH



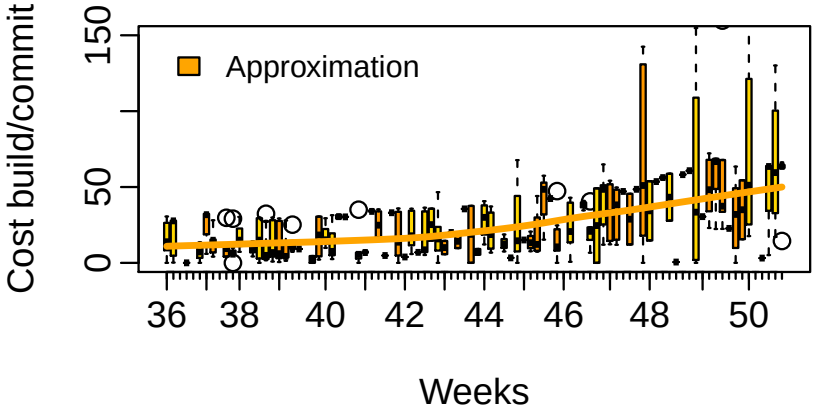
(a) Try branch



(b) Mozilla-inbound branch

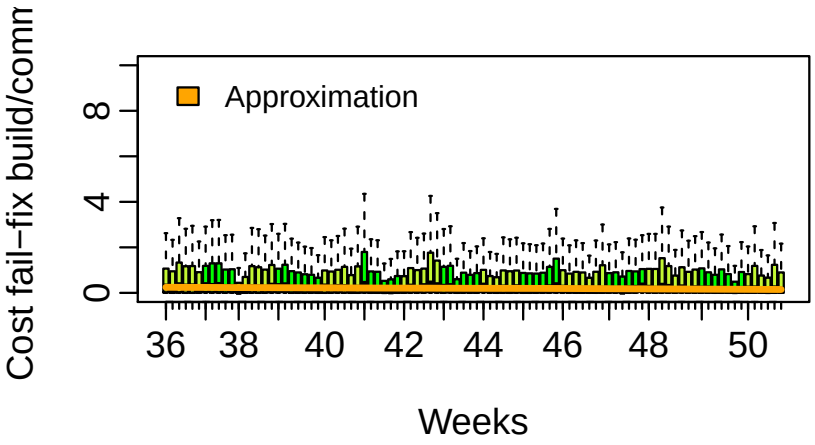


(c) Mozilla-central branch

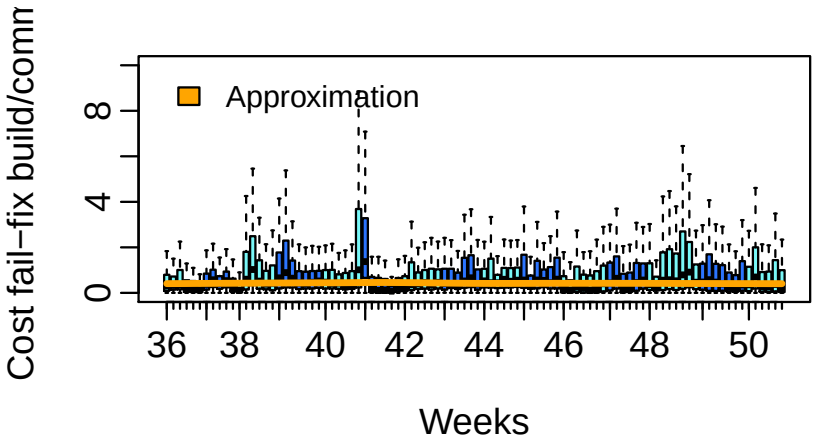


(d) Mozilla-release branch

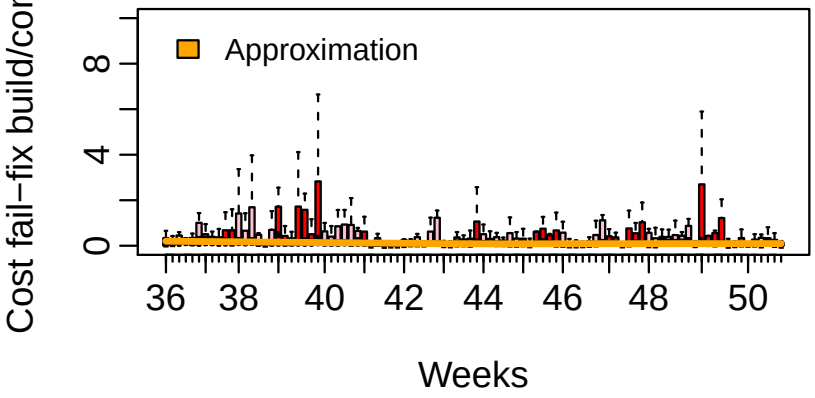
Figure B.1 Price (USD) by commit on each branch based on AWS' pricing.



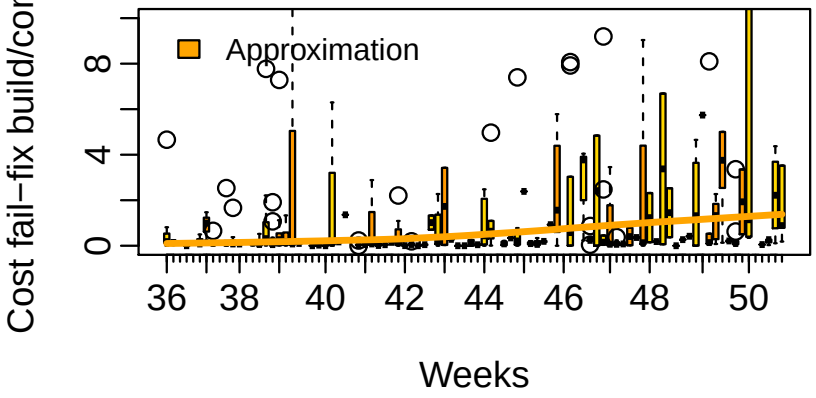
(a) Try branch



(b) Mozilla-inbound branch

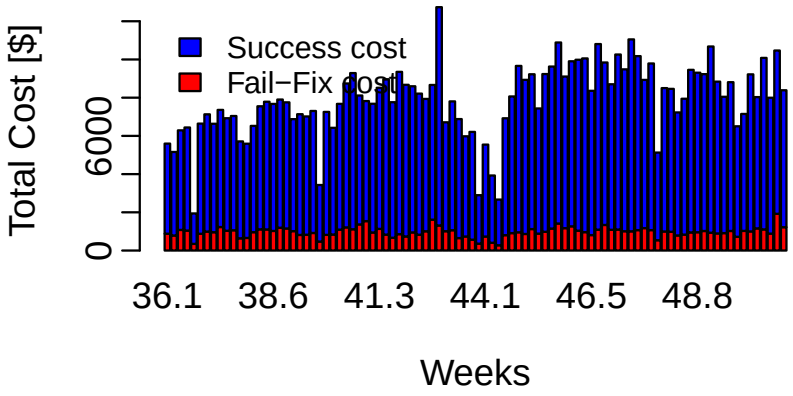


(c) Mozilla-central branch

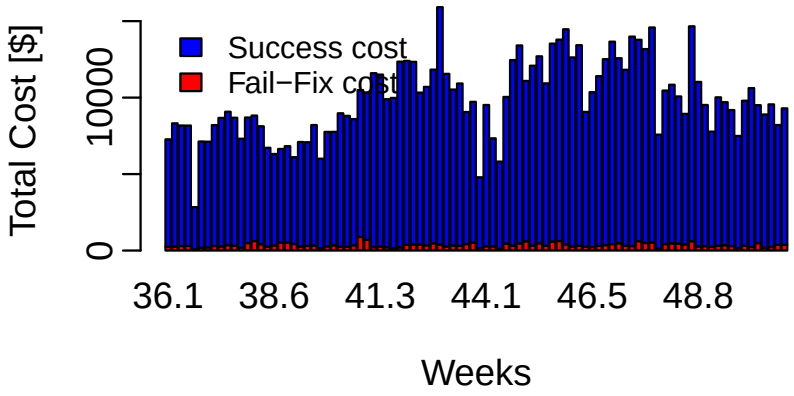


(d) Mozilla-release branch

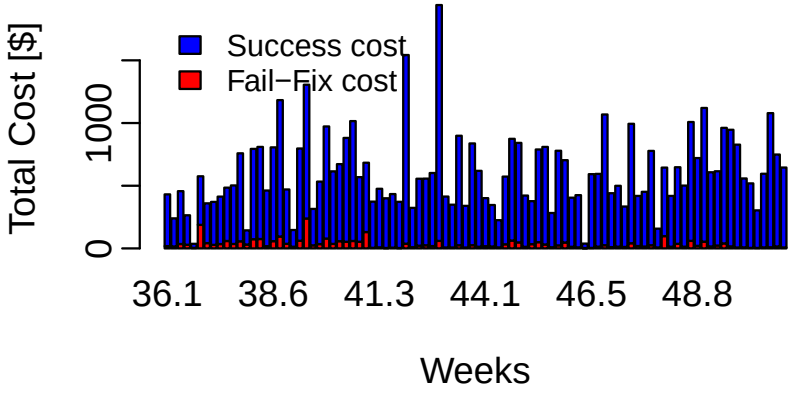
Figure B.2 Fix-fail price (USD) by commit on each branch based on AWS' pricing.



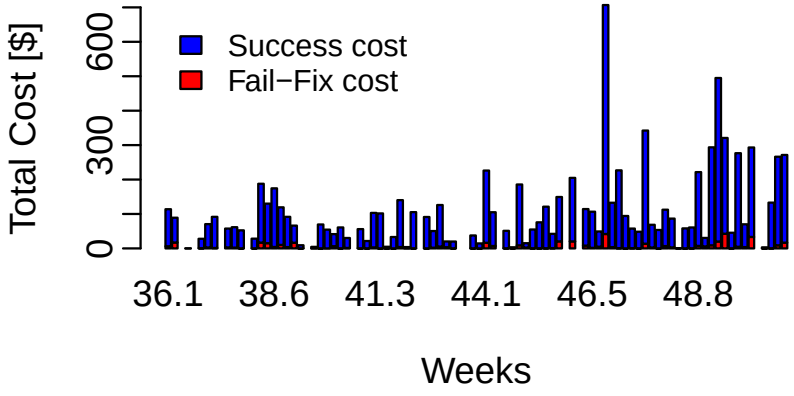
(a) Try branch



(b) Mozilla-inbound branch



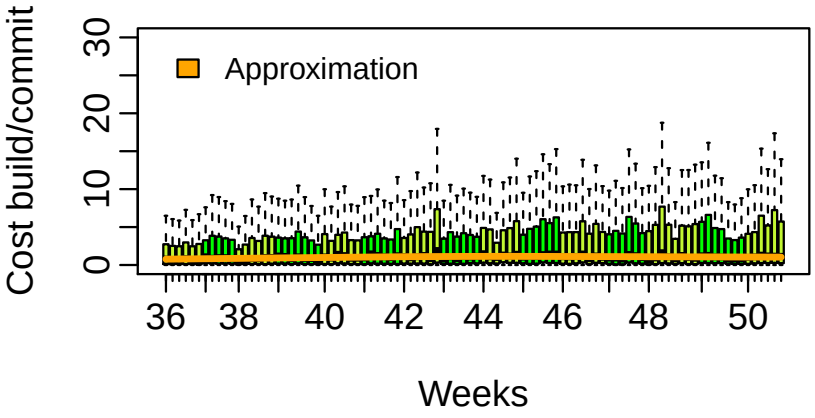
(c) Mozilla-central branch



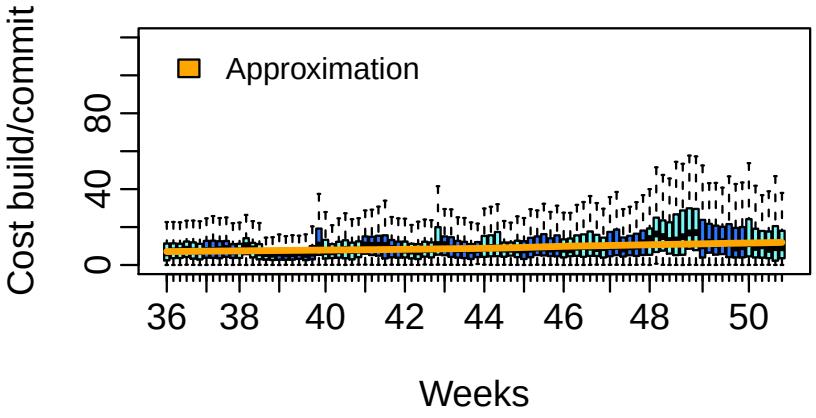
(d) Mozilla-release branch

Figure B.3 Total cost (USD) of builds by week on each branch based on AWS' pricing.

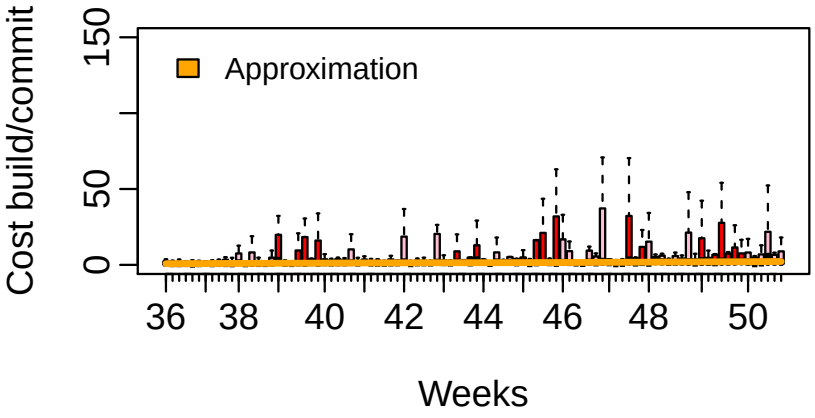
APPENDIX C COMPLEMENTARY RESULTS FOR RQ3, AZURE APPROACH



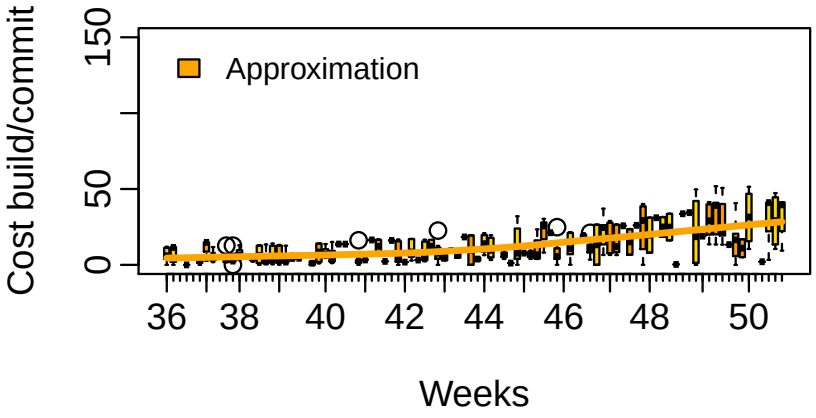
(a) Try branch



(b) Mozilla-inbound branch

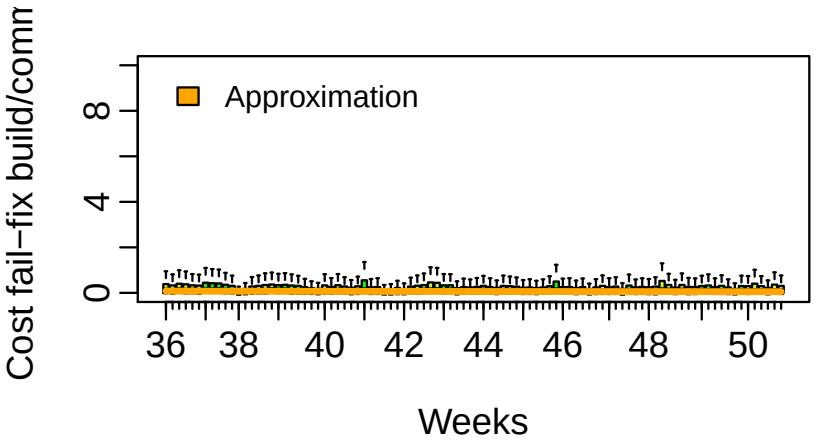


(c) Mozilla-central branch

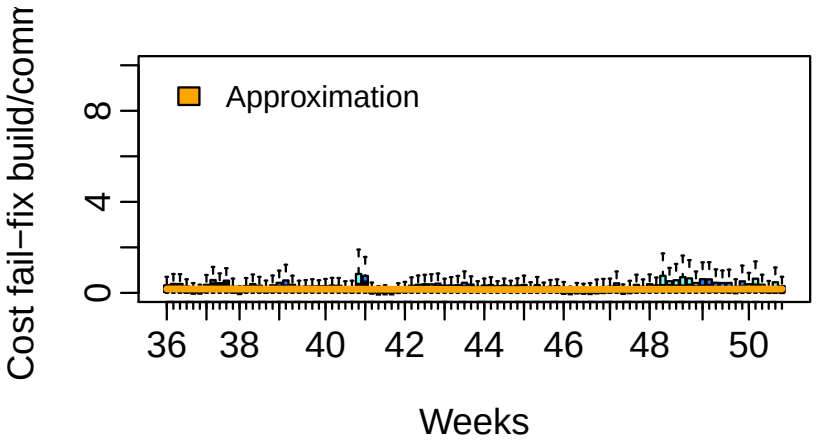


(d) Mozilla-release branch

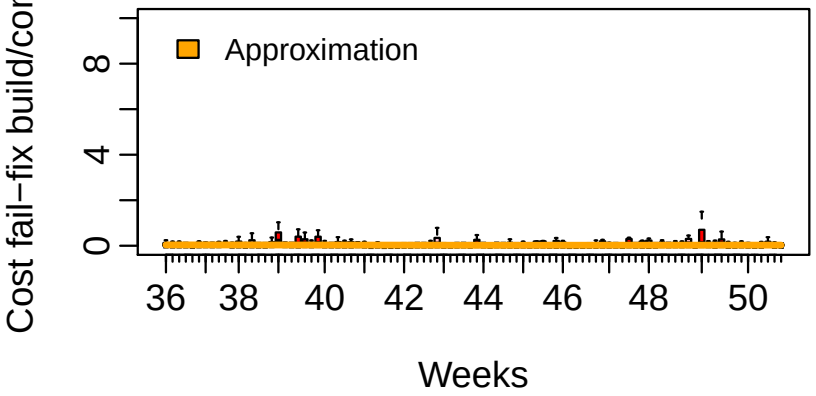
Figure C.1 Price (USD) by commit on each branch based on Azure's pricing.



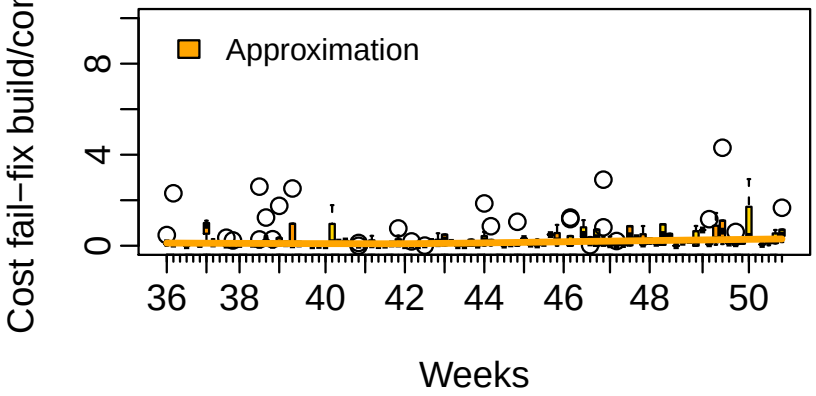
(a) Try branch



(b) Mozilla-inbound branch

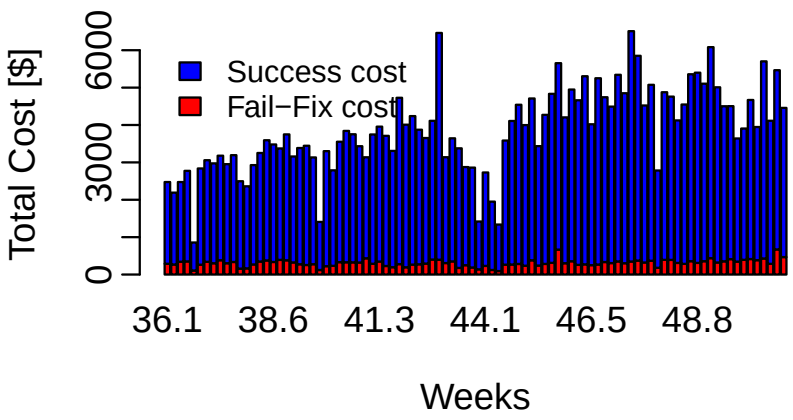


(c) Mozilla-central branch

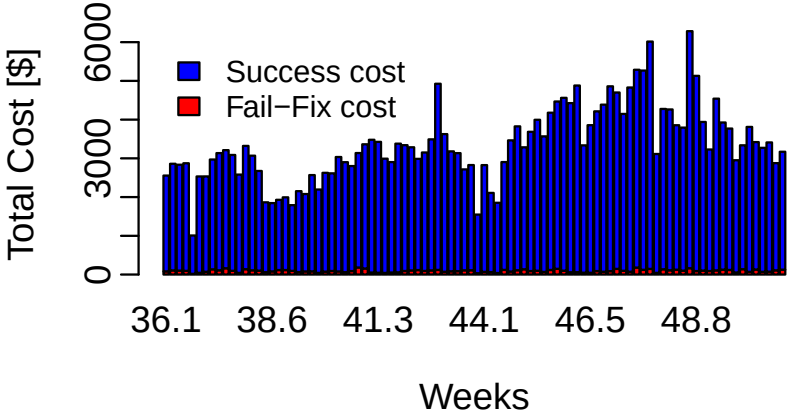


(d) Mozilla-release branch

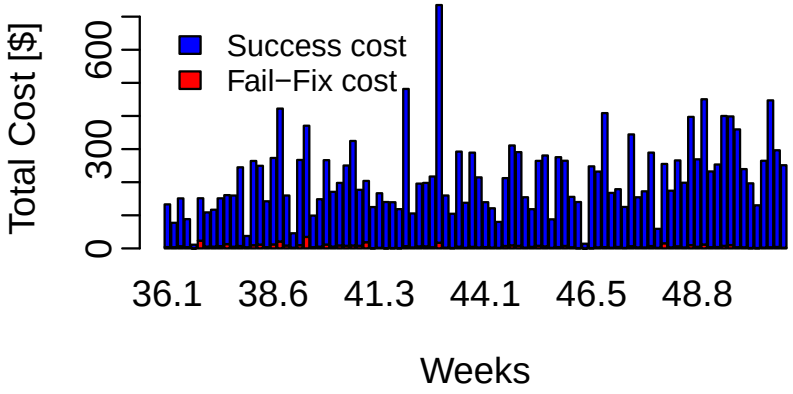
Figure C.2 Fix-fail price (USD) by commit on each branch based on Azure's pricing.



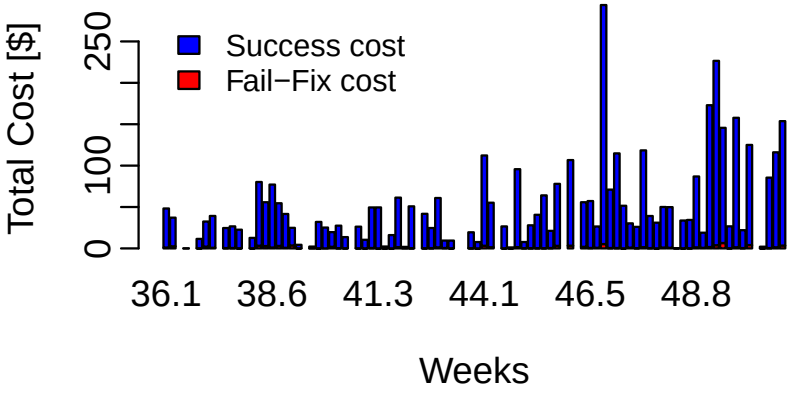
(a) Try branch



(b) Mozilla-inbound branch



(c) Mozilla-central branch



(d) Mozilla-release branch

Figure C.3 Total cost (USD) of builds by week on each branch based on Azure's pricing.

APPENDIX D EXAMPLE OF VOCABULARY REDUCTION FOR THE BROWN BUILD DETECTION METHODOLOGY.

Table D.1 Example of application of the six rules of vocabulary reduction.

Step	Log file
Original	Launching build on machine MyMachine CodeChange: A4E34FF1 from project my/project/path [2020-08-01 10:10] Run Test my_first_test ..... succeed [2020-08-01 10:12] Run Test my_second_test ..... succeed [2020-08-01 10:13] Run Test my_third_test ..... [error] connection issue: MyMachine not responding failed Build status: 1
Rule1	Launching build on machine MyMachine CodeChange: A4E34FF1 from project **rulepath** [2020-08-01 10:10] Run Test my_first_test ..... succeed [2020-08-01 10:12] Run Test my_second_test ..... succeed [2020-08-01 10:13] Run Test my_third_test ..... [error] connection issue: MyMachine not responding failed Build status: 1
Rule2	Launching build on machine MyMachine CodeChange: **ruleid** from project rulepath [2020-08-01 10:10] Run Test my_first_test ..... succeed [2020-08-01 10:12] Run Test my_second_test ..... succeed [2020-08-01 10:13] Run Test my_third_test ..... [error] connection issue: MyMachine not responding failed Build status: 1
Rule3	Launching build on machine MyMachine CodeChange ruleid from project rulepath Run Test my first test succeed Run Test my second test succeed Run Test my third test error connection issue MyMachine not responding failed Build status
Rule4	Launching build on machine My Machine **Code Change** ruleid from project rulepath Run Test my first test succeed Run Test my second test succeed Run Test my third test error connection issue **My Machine** not responding failed Build status
Rule5	Launching build machine Machine Code Change ruleid project rulepath Run Test first test succeed Run Test second test succeed Run Test third test error connection issue Machine responding failed Build status exit
Rule6	**launch** build machine machine code change ruleid project rulepath **r**un **t**est first test **succe** run test second test **succe** Run Test third test error **connec issu** machine **respond** fail **b**uild status

APPENDIX E EXAMPLE OF TF-IDF APPLICATION FOR THE BROWN BUILD DETECTION METHODOLOGY.

Table E.1 Example of application of TF-IDF.

Corpus	Log1 : "Test fail fail" Log2 : "Test fail pass" Log3 : "Test pass pass"
Original Matrix	$\begin{pmatrix} 1 & 0 & 2 \\ 1 & 1 & 1 \\ 1 & 2 & 0 \end{pmatrix}$ With each line representing a log and each column respectively the words "test", "pass", "fail".
TF	$\begin{pmatrix} 0.33 & 0 & 0.66 \\ 0.33 & 0.33 & 0.33 \\ 0.33 & 0.66 & 0. \end{pmatrix}$
IDF	$\begin{pmatrix} 0 & 0.41 & 0.41 \end{pmatrix}$
TF-IDF	$\begin{pmatrix} 0 & 0 & 0.27 \\ 0 & 0.14 & 0.14 \\ 0 & 0.27 & 0 \end{pmatrix}$

We can observe that the word "test" has the value 0 for all logs in the TF-IDF matrix since it is used in each log.