Titre: Title:	The Impact of Operating Systems and Environments on Build Results
Auteur: Author:	Mahdis Zolfagharinia
Date:	2017
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Zolfagharinia, M. (2017). The Impact of Operating Systems and Environments on Build Results [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/2861/

UNIVERSITÉ DE MONTRÉAL

THE IMPACT OF OPERATING SYSTEMS AND ENVIRONMENTS ON BUILD RESULTS

MAHDIS ZOLFAGHARINIA
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

THE IMPACT OF OPERATING SYSTEMS AND ENVIRONMENTS ON BUILD RESULTS

présenté par : ZOLFAGHARINIA Mahdis
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. QUINTERO Alejandro, Doctorat, président
M. ADAMS Bram, Doctorat, membre et directeur de recherche
M. GUÉHÉNEUC Yann-Gaël, Doctorat, membre et codirecteur de recherche
M. KHOMH Foutse, Ph. D., membre

DEDICATION

To my grandma,
Who always inspired me to achieve my goals

And to my parents,
Who always support and encourage me

ACKNOWLEDGEMENTS

I believe changing my program to a research based master was one of the best decisions for my academic life. It gave me the opportunity to meet new people and attend amazing conferences, which all are now part of this thesis. A especial thanks to my supervisor, Dr.Bram Adams for his kind guidance, great support and boundless patience all along the way. I learned a lot from working with a humble, talented and hard working supervisor like you.

I would also like to thank my co-supervisor Dr.Yann-Gaël Guéhéneuc for his kindness, motivation and immense knowledge. Thank you Yann for always being by my side, not just as a co-supervisor but as a mentor and a friend.

My sincere thanks to Dr.Foutse Khomh for accepting my invitation to be jury member and to Dr.Alejandro Quintero for accepting to be president of my defense.

I am grateful to my parents, and my siblings for their unconditional love. Without your support, this thesis would not have been possible.

I would like to thank all my colleagues in our lab and my friends : Amir, Parastou, Yujuan, Rodrigo, Bani, Alexandre, Ruben, Sepideh, Asana, and Antoine for all the moment we spent together, and all the memories we made.

I am grateful to the following university staff : Louise Longtin, Nathalie Audelin, Chantal Balthazard, and Brigitte Hayeur for their unfailing assistance.

Thanks for all your encouragement !

RÉSUMÉ

L'intégration continue (IC) est une pratique d'ingénierie logicielle permettant d'identifier et de corriger les fautes logicielles le plus rapidement possible après l'intégration d'un changement de code dans système de contrôle de versions. L'objectif principal de l'IC est d'informer les développeurs des conséquences des changements effectués dans le code. L'IC s'appuie sur différents systèmes d'exploitation et environnements d'exécution pour vérifier si un système fonctionne toujours après l'intégration des changements. Ainsi, de nombreux "builds" sont créés, alors que seulement quelques-uns révèlent de nouvelles fautes. En d'autres termes, un phénomène d'inflation des builds se produit, où le nombre croissant de builds a un rendement décroissant. Cette inflation rend l'interprétation des résultats des builds difficile, car l'inflation augmente l'importance de certaines fautes, alors qu'elle cache l'importance d'autres. Cette thèse fait progresser notre compréhension de l'impact des systèmes d'exploitation et des environnements d'exécution sur les fautes des builds et le biais potentiel encouru à cause de l'inflation des builds par une étude à grande échelle de 30 millions de builds de l'écosystème CPAN. Nous choisissons CPAN parce que CPAN fournit un riche ensemble de données pour l'analyse automatisée des builds sur des douzaines d'environnements (versions de Perl) et systèmes d'exploitation. Cette thèse rapporte une analyse quantitative et qualitative sur les fautes dans les builds pour classer ces fautes et trouver la raison de leur apparition. Nous observons : (1) l'évolution des fautes des builds au fil du temps et rapportons que plus de builds sont effectués, plus le pourcentage de fautes de builds diminue, (2) différents environnements et systèmes d'exploitation mettent en avant différentes fautes, (3) les résultats des builds doivent être filtrés pour identifier des fautes fiables, et (4) la plupart des fautes des builds sont dus à leur dépendance à l'API. Les chercheurs et les praticiens devraient tenir compte de l'impact de l'inflation des builds lorsqu'ils analysent ou exécutent des builds.

ABSTRACT

Continuous Integration (CI) is a software engineering practice to identify and correct a defect as soon as possible after a code change has been integrated into the version control system. The main purpose of CI is to give developers a quick feedback of code changes. These changes build on different OSes and runtime environments to check backward compatibility as well as to check if the product still works with the new changes. So, many builds are performed, while only a few of them can identify new failures. In other words, a phenomenon of build inflation can be observed, where the increasing number of builds has diminishing returns in terms of identified failures vs. costs of running the builds. This inflation makes interpreting build results challenging as it increases the importance of some failures, while it hides the importance of others. This thesis advances our understanding of the impact of OSes and runtime environments on build failures and build inflation through a large-scale study of 30 million builds of the CPAN ecosystem. We choose CPAN because CPAN provides a rich data set for the analysis of automated builds on dozens of environments (Perl versions) and operating systems.

This thesis performs quantitative and qualitative analysis on build failures to classify these failures and find out the reason of their occurrence. We observe: (1) the evolution of build failures over time and report that while more builds are being performed, the percentage of them identifying a failure drops, (2) different OSes and environments are not equally reliable, (3) the build results of CI must be filtered to identify reliable failing data, (4) and most build failures are due to API dependency. Researchers and practitioners should consider the impact of build inflation when they are analyzing and-or performing builds.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

OS	Operating System
CI	Continuous Integration
VCS	Version Control System
DevOps	Development and Operations
POSIX	Portable Operating System Interface
API	Application Programming Interface
MSR	Mining Software Repository
POM	Project Object Model
PBP	Perl Best Practice
AWS	Amazon Web Services
QA	Quality Assurance

CHAPTER 1 INTRODUCTION

Continuous integration (CI) automates the compilation, building, and testing of software. A CI build server automates the build process of a software project so that it is built as soon as developers make any changes, making sure to compile, test and analyze the impact of those changes on the system. Each new commit entered into the VCS trigger the CI tool (e.g., Jenkins), which compiles and tests the project partially or fully. The main purpose of CI is to help developers to detect build failures as soon as possible [1] to reduce the risk of defective releases. In the last decade, CI has become popular in both industrial and open-source software (OSS) community [2, 3]. Automation of CI can help increase the speed of project development and expose potential failures as soon as possible. Many well known companies practice CI such as Google, Facebook, Linkedin and Netflix [4].

Build systems automate the process of compiling a software project into executable artifacts rapidly across a variety of operating systems and programming languages [5] by using compilers, scripts and other tools. They play a crucial role in the software development process. Developers rely on the build systems to test their modifications to the source code, while testers use the build systems for executing automated tests to check if the expected output is still be provided after changes. Many researchers have studied build systems [6, 7, 8].

Initially, developers wrote ad hoc programs to do the whole compilation and test, until the build language **-Make-** was designed at Bell labs in 1979 by Feldman as a build automation tool [9]. It became the initial build system for Unix-like systems. Whenever a part of the program is changed, Make performs just the necessary commands to recompile the related files to create an up-to-date executable. Make does this by finding the name of a required target in the expression, assuring all of the dependent files exist, and then creating the target. Table 1.1 shows an example of a Make target expression. Make represents dependencies among files in a directed acyclic graph (DAG) The DAG can vary among build executions, different tools, and configurable features. When a change happens, the DAG informs Make about what has to be rebuilt.

As different programming languages gained popularity, different build automation systems

Table 1.1 Target expression in a makefile

Makefile Expression	Example
target: dependencies command(s)	main.o: main.c text.h cc -c main.c

were developed. Unix-like systems mainly use make-based tools such as GNU make and BSD make. Ruby based (Rake), LISP-based (ASDF), C#-based (Cake), and Java-based (Ant, Maven, and Jenkins) systems use Non-Make-Based tools. Finally, Facebook's Buck and Google's Blaze are other recent build system.

As other example, **Maven** uses build-by-convention rather than a complicated build language, While other build systems like Ant needs developers to write all the commands. Maven is based on the core concept of a build lifecycle, which refers to an explicit, standardized process for building each code or other artifact. The developer building a project must use a small set of commands to compile a Maven project, then the instructions mentioned in the pom.xml file will assure the proper outputs. Furthermore, dependency management is a mechanism to centralize API dependencies using an online repository to hold build artifacts and to specify constraints to control the versions of API libraries to be used for building the software project.

Testing is a critical phase of the build process to guarantee software quality. The build system runs the compiled software project through a series of automated tests [10]. After compiling the new code changes, the CI system will execute the compiled system using a variety of test types such as unit test, component tests, and regression testings to assure that the product works properly [11]. Developers often run other types of tests, such as performance testing to check if all performance requirements have been met [8].

Build Inflation is a phenomenon, which is produced due to the massive builds at the CI practice. In theory, all additional builds/tests add more information, because a failed run clearly identifies a problem with the new code changes, while a successful run allows to eliminate a certain build from the list of suspicious builds. However, in practice, some builds/tests add more information than others. Each code change typically is built on multiple OSes, runtime environments and hardware architectures to check if a software product works on that particular OS or environment. Therefore, for every new commit/release, we have large numbers of builds. Figure 2.3 shows how one commit in Treeherder was built on 54 different OSes, and just for the first OS, there are 74 different test suites. While every new build provides new information, the number of builds across environments and OSes is not homogeneously distributed, which gives more weight to some failures than others. Therefore, even though, each additional build brings new information, these builds have different values and are not equally informative.

It is essential to reduce build inflation for several reasons. First, each build by itself needs noticeable time and effort for executing. As such, scheduling additional builds only increases this time and effort, but without corresponding gains in QA. Apart from time lost, there are also monetary losses associated with redundant builds. Although cloud services, e.g., AWS or Microsoft Azure, etc., provide a faster and cheaper build infrastructure, they do have a non-negligible cost. O'Duinn, the former release manager of Amazon, discussed about the financial cost of a checkin in his blog [12]. The Amazon prices for the two cheapest AWS regions (US-west-2 and US-east-1) for daily load on an OnDemand builder cost $0.45, and OnDemand tester costs $0.12 per hour. He mentioned that a checkin costs at least USD$30.60 to answer "how much did that checkin actually cost Mozilla". The cost included the following usage: USD$11.93 for Firefox builds/tests, USD$5.31 for Fennec builds/tests and USD$13.36 for B2G builds/tests. Thus, we can define the approximate cost that we can save by deleting extra/unnecessary builds. We can decide using this information if developer time is worth spending on scheduling for the valuable builds to avoid all these costs. Moreover, as O'Duinn has mentioned, "network traffic is still a valid concern, both for reliability and for cost". Hence, even in the presence of large-scale cloud infrastructure, awareness and reduction of build inflation is a must.

1.1 Research Hypothesis: Build Inflation Consequences

This thesis aims to study the impact of build inflation with the following hypothesis:

> Build inflation is an actual phenomenon in open-source projects that impacts the interpretation of build results.

To explore this hypothesis we analyse the impact of "build inflation" to study the returns of each additional build. Bias in build outcomes can be introduced due to large number of builds on diverse combinations of OSes and environments. This bias makes it complicated to interpret build outcomes. Practitioners and researchers should consider it while investigating build outcomes. The following section describes my contribution and its connections with the research hypothesis.

1.2 Thesis Contributions: The Impact of OSes/Environments on Build Inflation

Just as Maven for the Java programming language, CPAN (Comprehensive Perl Archive Network) [13] is an ecosystem of modules (APIs/libraries) for Perl. CPAN has its own CI system

that builds and tests any new release of a module on a divers range of operating systems and run-time environments (Perl versions). Similar to Maven and npm, but different from TravisCI and Jenkins, CPAN's CI works at release-level not at commit-level.

We selected CPAN in this study as it provides a rich data set for the analysis of builds and build inflation across different OSes and run-time environments. We performed quantitative and qualitative analysis on build failures to understand the impact of OS and run-time environment on 30 million builds of the CPAN ecosystem. We also studied the evolution of build failures over time, which allowed us to study build inflation. We mined the log files and meta-data of all CPAN modules to categorize build failures and the cause of their occurrences (faults). We studied 30 million builds in 5.5 years for 12,584 module versions, 27 OSes, and 103 (Perl) environments.

Our first contribution is the finding that the build failure ratio across all modules decreases over time, while the number of builds per module increases about 10 times. Moreover, the build results on different OSes/environments are not equally reliable, not only due to the fewer number of builds in some OSes, e.g., 40% of builds in Linux vs. less than 5% in Windows, but also due to the lower popularity of some OSes among Perl developers. Our findings show that 86.5% of build results consistently succeed or fail across all OSes/environments, which is an evidence of build inflation. Indeed, if a certain build failure, which consistently succeeds/fails everywhere, has (not) been found for one OS or environment build, remaining builds are no longer necessary from the perspective of that build failure. Our findings also show that failures occurring only on a minority of operating systems usually occur on the operating systems that are the least supported by the developers (Windows in the case of CPAN). As Linux and Freebsd are the main OSes for CPAN, so we should treat the build failures on these two OSes as high priority. These findings showed different builds have different values and must be prioritized to avoid the consequence of build inflation that can increase the importance of some failures while it hides the importance of others. We categorized build failures into six main categories and nine subcategories. We showed that API dependency is the highest reason of build failures either in minority or in majority failing builds. We also showed that programming issues are the 2nd highest reason of failure when majority of OSes fail, while configuration issues are the 2nd highest reason when minority of OSes fail.

1.3 Organization of Thesis

This thesis is structured as follows: Chapter 2 discusses prior research related to our work. Chapter 3 presents the research process and the organization of the thesis. Chapter 4 introduces the comprehensive structure and details of our empirical studies, which is my accepted paper at MSR 2017, followed by general discussion in chapter 5, and then conclusion and future work in Chapter 6.

CHAPTER 2 LITERATURE REVIEW

In this chapter, we introduce prior research on CI and discuss the studies most relevant to this thesis.

2.1 State-of-the-practice

CI is vital for modern software development. A CI tools like Jenkins executes builds several times a day for patches under review, and for each new commit entering into the VCS, to send an early notification to developers about any build failure [14, 15].
Continuous integration is done both at release-level (consider official releases of a project) and at commit-level (consider each new changes in the source code). Some of popular CI tools are Travis CI [16], Strider [17], Jenkins [18], TeamCity [19], Hudson, and Go. We briefly present some of these CI tools. Companies use the most appropriate tools based on the features they need. For example, the Eclipse community adopted Jenkins, while Mozilla and Google Chromium use Buildbot [20], which has server-slave-based structure, and can be customized based on demands.

2.1.1 BuildBot

Buildbot is adopted by Mozilla, Chromium, and WebKit. Buildbot is a Python-based tool, and can be deployed on POSIX-compliant operating systems [21]. Buildbot is a job scheduling system, and it executes the jobs whenever resources are available. Buildbot consists of one or more build masters and many workers, as soon as the masters monitor changes, workers run builds on a variety of operating systems. Figure 2.1 presents build results in Buildbot dashboard across different operating systems.

2.1.2 Jenkins

Jenkins is an open-source CI tool that is popular among DevOps for being free, open-source and modular (with over 1K plugins) [21]. Figure 2.2 shows the Jenkins dashboard, which illustrates the build history of new commits.

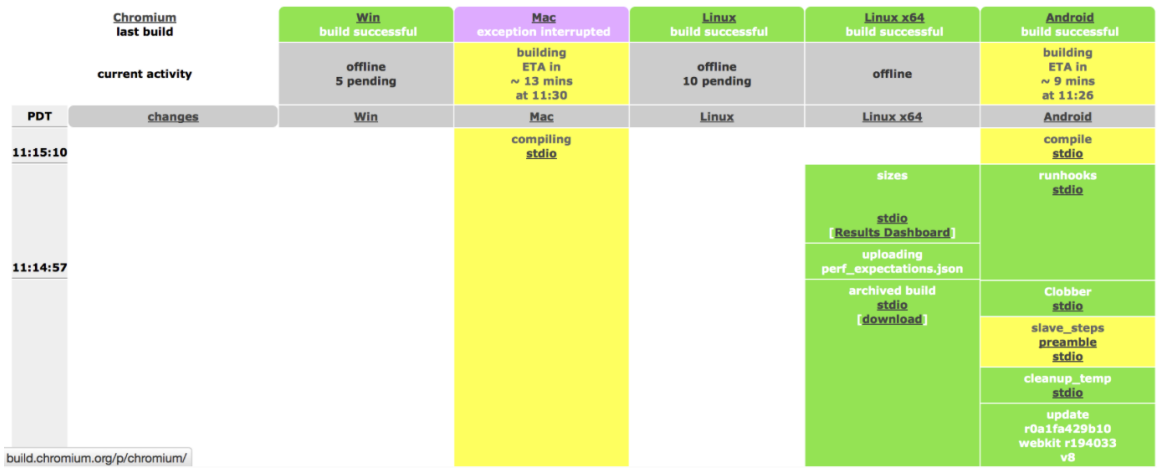

Figure 2.1 BuildBot

Figure 2.2 Jenkins CI tool

2.1.3 Travis CI

Travis CI is a hosted CI service, which is integrated with GitHub. Travis CI's build environment provides several run-time environments for multiple programming languages, e.g., Ruby, PHP, Node.js. While Travis CI repository is hosted on GitHub and can be setup quickly, Jenkins needs to be hosted, configured and installed.

2.1.4 Treeherder

Mozilla uses Treeherder as a reporting dashboard for commit-level CI results of its projects [22]. Treeherder also has a rich set of APIs that can be adapted in other projects to provide the required information. Figure 2.3 shows the results of automated builds and the related tests for only one commit on Treeherder [23], that were performed on November 3, 2017. It shows build results on 54(!) combinations of build configurations and operating systems (multiple versions of Linux, OS X, Windows, Android). A build configuration is a specific selection of features (e.g., QuantumRender or Stylo) and build tool parameters (e.g., opt and debug). Figure 2.3 also reports the results of test suites for each of the 54 builds. With different tests being run on different operating systems, build results are not straightforward to interpret for outsiders.

It seems that an explosion in builds is happening, and that there is an inflation in builds, i.e., many builds are happening, most of them succeeding, and very few number of builds fail, which lead to diminishing returns. It means that too many builds are happening for each new commit/release, while only few of them fail, i.e., for the first OS among 74 builds, only one failure happened.

2.1.5 CPAN

CPAN presents a rich data set of the results of automated builds for the Perl programming language on a variety of operating systems and run time environments. Figure 4.1 shows the build dashboard for one particular Perl package, providing a summary of build results across different operating systems and run-time environments. CPAN's CI works at package release-level like Maven, not at commit-level like Travis CI and Jenkins. Release-level CI is the process of scheduling and controlling a software build via a variety of OSes and environments, which consists of testing and deploying software releases, and consider official releases of a project to get this the level of granularity instead of considering each new change on the code.

Linux opt	tc(B Cpp GTest Jit1 Jit2 Jit3 Jit4 Jit5 Jit6) tc-Fxfn-l-e10s(en-US) tc-Fxfn-r-e10s[tier 2](en-US) tc-M(a11y c1 c2 c3) tc-M-e10s(+28) tc-R-e10s (+20) tc-W-e10s(wpt12* +17) tc-X(+8) tc-e10s(Mn)
Linux pgo	tc(B)
Linux debug	SM-tc(arm) tc(B Cpp GTest Jit1 Jit2 Jit3 Jit4 Jit5 Jit6) tc-Fxfn-l-e10s(en-US) tc-Fxfn-r-e10s[tier 2](en-US) tc-M(+4) tc-M-e10s(1 +40) tc-R-e10s (+20) tc-W-e10s(+18) tc-X(X1 X2 X3 X4 X5 X6 X7 X8 X9 X10 X11 X12 +18) tc-e10s(Mn)
Linux Stylo Disabled opt	tc(Cpp) tc-M(+4) tc-M-e10s(+20) tc-R-e10s(C R1 R2 R3 R4 R5 R6 R7 R8 +20) tc-SYsd-e10s(sy) tc-W-e10s(+18)
Linux Stylo Disabled debug	tc(Cpp) tc-M(a11y c1 c2 c3) tc-M-e10s(+40) tc-R-e10s(C R1 R2 R3 R4 R5 R6 R7 R8 +40) tc-W-e10s(+18)
Linux x64 opt	S SM-tc(+5) SM-tc[tier 2](rust) T-e10s(+16) Tss-e10s(tp6) tc(+11) tc-Fxfn-l-e10s(en-US) tc-Fxfn-r-e10s[tier 2](en-US) tc-M(+4) tc-M-e10s (+33) tc-R-e10s(+28) tc-SY-e10s(sy) tc-W-e10s(+19) tc-X(X1 X2 X3 X4 X5 X6 X7 X8 +19) tc-e10s(+2) tc-e10s[tier 2](TV) [tier 2](AB)
Linux x64 pgo	T-e10s(+16) Tss-e10s(tp6) tc(+9) tc-Fxfn-l-e10s(en-US) tc-Fxfn-r-e10s[tier 2](en-US) tc-M(+4) tc-M-e10s(bc3* bc4* +26) tc-R-e10s(+20) tc-W-e10s(Wd Wr1 Wr2 Wr3 Wr4 Wr5 Wr6 wpt1 wpt2 wpt3 wpt4 wpt5 wpt6 wpt7 wpt8 wpt9 wpt10 wpt11 wpt12 +20) tc-X(+8) tc-e10s(+2) tc-e10s[tier 2](TV)
Linux x64 asan	tc(Bd Bo BoR Bof Cpp GTest Jit1 Jit2 Jit3 Jit4 Jit5 Jit6) tc-Fxfn-l-e10s(en-US) tc-Fxfn-r-e10s[tier 2](en-US) tc-M(c1* +3) tc-M-e10s(mda2* +43) tc-R-e10s(C J1 J2 J3 R1 R2 R3 R4 R5 R6 R7 R8 Ru1 Ru2 Ru3 Ru4 Ru5 Ru6 Ru7 Ru8 +43) tc-X(+8) tc-e10s(MnH* Mn) tc-e10s[tier 2](TV)
Linux x64 debug	S SM-tc(+7) tc(+11) tc-Fxfn-l-e10s(en-US) tc-Fxfn-r-e10s[tier 2](en-US) tc-M(+4) tc-M-e10s(16* +63) tc-R-e10s(+28) tc-W-e10s(+19) tc-X(X1 X2 X3 X4 X5 X6 X7 X8 X9 X10 +19) tc-e10s(Mn MnH +19) tc-e10s[tier 2](TV)
Linux x64 QuantumRender opt	tc-M-e10s(gl1 gl2 gl3 gpu mda1 mda2 mda3) tc-R-e10s(C J1 J2 J3 J4 R1 R2 R3 R4 R5 R6 R7 R8)
Linux x64 QuantumRender debug	tc-M-e10s(gl1 gl2 gl3 gpu mda1 mda2 mda3*) tc-R-e10s(C J1 J2 J3 J4 R1 R2 R3 R4 R5 R6 R7 R8)
Linux x64 Stylo Disabled opt	tc(Cpp) tc-M(+4) tc-M-e10s(+28) tc-R-e10s(C R1 R2 R3 R4 R5 R6 R7 R8 +28) tc-SYsd-e10s(sy) tc-W-e10s(+18)
Linux x64 Stylo Disabled debug	tc(Cpp) tc-M(+4) tc-M-e10s(+48) tc-R-e10s(C R1 R2 R3* R4 R5 R6 R7 R8) tc-W-e10s(+18)
Linux x64 Stylo-Seq opt	tc-SYss-e10s[tier 2](sy)
Linux x64 NoOpt debug	tc[tier 2](B)
OS X 10.10 opt	tc(+7) tc-Fxfn-l-e10s(en-US) tc-Fxfn-r-e10s[tier 2](en-US) tc-M(+4) tc-M-e10s(bc2* +25) tc-R-e10s(+4) tc-SY-e10s(sy) tc-T-e10s(g2* g1 +15) tc-W-e10s(+6) tc-X(X) tc-e10s(+2) tc-e10s[tier 2](TV)
OS X 10.10 debug	tc(+3) tc-Fxfn-l-e10s(en-US) tc-Fxfn-r-e10s[tier 2](en-US) tc-M(+4) tc-M-e10s(1* +25) tc-R-e10s(+5) tc-W-e10s(+11) tc-X(X) tc-e10s[tier 2](
OS X Cross Compiled opt	tc(B)
OS X Cross Compiled debug	tc(B)
OS X Cross Compiled NoOpt debug	tc[tier 2](B)
OS X 10.10 Stylo Disabled opt	tc(Cpp) tc-M(+4) tc-M-e10s(1 2 3 4 5 bc1 bc2 bc3 bc4 bc5 bc6 bc7 cl gl1 gl2 gl3 gpu mda +4) tc-R-e10s(+2) tc-SYsd-e10s(sy) tc-W-e10s(Wr wpt1 wpt2 wpt3 wpt4 wpt5 wpt6 wpt7 wpt8 wpt9 wpt10 wpt11 wpt12)
OS X 10.10 Stylo Disabled debug	tc(Cpp) tc-M(+4) tc-M-e10s(mda* +17) tc-R-e10s(+3) tc-W-e10s(Wr wpt1 wpt2 wpt3 wpt4 wpt5 wpt6 wpt7 wpt8 wpt9 wpt10 wpt11 wpt12 +3)
Windows 7 pgo	M-e10s(cl) T-e10s(+16) Tss-e10s(tp6) tc(+2) tc-Fxfn-l-e10s(en-US) tc-Fxfn-r-e10s[tier 2](en-US) tc-M(a11y c1 c2 c3) tc-M-e10s(2* +31) tc-R-e10s(+99) tc-SY-e10s(sy) tc-W-e10s(+13) tc-X(X*) tc-e10s(+2) tc-e10s[tier 2](TV)
Windows 7 debug	M(cl) M-e10s(cl) tc(+3) tc-Fxfn-l-e10s(en-US) tc-Fxfn-r-e10s[tier 2](en-US) tc-M(c3* +32) tc-M-e10s(+32) tc-R-e10s(+27) tc-W(+13) tc-W-e10s(+13) tc-X(X) tc-e10s(+2) tc-e10s[tier 2](TV)
Windows 7 Stylo Disabled debug	M-e10s(cl) tc(Cpp) tc-M(a11y c1 c2 c3) tc-M-e10s(+24) tc-R-e10s(C R1 R2 R3 R4 R5 R6 R7 R8 +24) tc-W-e10s(+13)
Windows 10 x64 opt	M-e10s(cl) R-e10s(+2) T-e10s(+15) Tss-e10s(tp6) tc(Cpp GTest Jit) tc-Fxfn-l-e10s(en-US) tc-Fxfn-r-e10s[tier 2](en-US) tc-M(a11y c1 c2 c3) tc-M-e10s(bc2* bc2* bc2* +34) tc-R-e10s(C J1 J2 +34) tc-SY-e10s(sy) tc-W-e10s(+13) tc-X(X*) tc-e10s(+2) tc-e10s[tier 2](TV)
Windows 10 x64 pgo	M-e10s(cl) R-e10s(R-e10s Ru) T-e10s(+15) Tss-e10s(tp6) tc(+2) tc-Fxfn-l-e10s(en-US) tc-Fxfn-r-e10s[tier 2](en-US) tc-M(c1* c3* +2) tc-M-e10s(bc2* bc2* bc2* +29) tc-R-e10s(C J1 J2 +29) tc-SY-e10s(sy) tc-W-e10s(+13) tc-X(X) tc-e10s(+2) tc-e10s[tier 2](TV)
Windows 10 x64 debug	M-e10s(cl) R(+4) R-e10s(+4) tc(+3) tc-Fxfn-l-e10s(en-US) tc-Fxfn-r-e10s[tier 2](en-US) tc-M(+4) tc-M-e10s(bc5* dt7* +33) tc-R(+3) tc-R-e10s(+3) tc-W-e10s(Wr wpt1 wpt2 wpt3 wpt4 wpt5 wpt6 wpt7 wpt8 wpt9 wpt10 wpt11 wpt12 +3) tc-X(X) tc-e10s(Mn* MnH) tc-e10s[tier 2](TV)
Windows 10 x64 Stylo Disabled opt	M-e10s(cl) R-e10s(R-e10s) tc(Cpp) tc-M(c1* +3) tc-M-e10s(bc2* bc2* bc2* +21) tc-R-e10s(C) tc-SYsd-e10s(sy*) tc-W-e10s(+13)
Windows 10 x64 Stylo Disabled debug	M-e10s(cl) R-e10s(R-e10s1 R-e10s2) tc(Cpp) tc-M(a11y c1 c2 c3) tc-M-e10s(gl8* +22) tc-R-e10s(C) tc-W-e10s(+13)
Windows 2012 opt	SM-tc(p) tc(B* B* B S)
Windows 2012 pgo	tc(B Bs)
Windows 2012 debug	SM-tc(+2) tc(+3)
Windows 2012 NoOpt debug	tc[tier 2](B)
Windows 2012 x64 opt	tc(B Bs S)
Windows 2012 x64 pgo	tc(B Bs)
Windows 2012 x64 debug	tc(B Bs S)
Windows 2012 x64 NoOpt debug	tc[tier 2](B)
Android 4.0 API16+ opt	tc(B)
Android 4.0 API16+ debug	tc(B)
Android 4.2 x86 opt	tc(B gv) tc-M(c1 c2) tc-X(X1 X2 X3 X4 X5 X6)
Android 4.3 API16+ opt	tc(+2) tc-M(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 c1 c2 cl gpu mda1 mda2 mda3 rc1 rc2 rc3 rc4) tc-R(C1 C2 C3 C4 J1 J2 J3 J4 J5 J6 J7 J8 J9 J10 J11 J12 J13 J14 J15 J16 J17 J18 J19 J20 J21 J22 J23 J24 J25 J26 J27 J28 J29 J30 J31 J32 J33 J34 J35 J36 J37 J38 J39 J40 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R11 R12 R13 R14 R15 R16 R17 R18 R19 R20 R21 R22 R23 R24) tc-X(X1 X2 X3 X4 X5 X6 X7 X8)
Android 4.3 API16+ debug	tc[tier 2](Mn1 Mn2 Mn3 Mn4 Mn5 Mn6 Mn7 Mn8 Mn9 Mn10) tc(Cpp gv) tc-M(1 2* 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 36 37 38 39 40 41 42 43 44 45 46 47 48 c1 c2 c3 c4 cl gpu mda1 mda2 mda3) tc-R(+158) tc-X(X1 X2 X3 X4 X5 X6 X7 X8 +158)
Android 5.0 AArch64 opt	tc(B)
Android API16+ Gradle opt	tc(Bng) tc-M[tier 2](c1 c2 rc1 rc2 rc3 rc4)
Gecko Decision Task opt	D
Linting opt	Bugzilla
windows2012-32-rusttests opt	tc[tier 2](BR*)
linux64-rusttests debug	tc[tier 2](BR)
windows2012-64-rusttests opt	tc[tier 2](BR)
linux32-rusttests opt	tc[tier 2](BR)
windows-mingw32-32 debug	tc[tier 2](B)
linux32-rusttests debug	tc[tier 2](BR)
linux64-rusttests opt	tc[tier 2](BR)

Figure 2.3 TreeHerder CI tool dashboard

2.2 State-of-the-art

Continuous Integration focuses on integrating code changes by several developers constantly, avoiding unpredictable integration close to release. Developers get a quicker feedback on the changes with this automation process in CI [15]. A CI server monitors the version control system (VCS), builds the latest code snapshot, then runs tests. Therefore, if a new commit or release is available in the VCS, build and test systems are provoked to compile and test.

Seo et al. [5] concentrated on the compiler errors that occur in Google's build process for Java and C++ environments. They assessed 26.6 million builds -and found that- the proportion of failures for C++ and Java builds is 37.4% and 29.7%. They presented multiple patterns of build failures and concluded that API dependencies are the major reason of compilation errors. They mentioned that the time to fix the failures varies greatly and different tools are required to support developers' needs. We complement this study with a larger-scale study of 68.9 million builds of CPAN that shows that build failures due to programming issues occur with a ratio of 36.1% that happen across most of the operating systems, while it reduces to 15.5% when a failure impacts only a few operating systems. Also, we analyzed build failures for Perl, which is an interpreted language, while Seo et al. studied compiled languages (Java and C++).

Vasilescu et al. [24] conducted a preliminary study on 246 GitHub projects to investigate the impact of adopting CI systems (such as Travis CI) on productivity and software quality. They declared that Continuous Integration usage can increase productivity but it does not necessarily increase software quality. Also, they introduced evidence on the advantages of CI, they did not discuss the consequences of performing many builds in CI. This thesis is analyzing the impact of adopting CI on build results considering factors such as operating systems and environments within the CPAN CI system. Armed with this understanding and increasing demand for CI usage, researchers and developers must be careful about using CI to avoid build inflation and its consequences.

Researchers studied TRAVIS CI as a data source [3]. They showed that CI services (CIS) like Travis CI [16], which is a globally accessible CIS, were widely used and increased the developers' productivity. They analyzed 34,544 open-source projects from GitHub and found that over 40% of these projects use CI. They analyzed almost 1.5 million builds of Travis CI to understand the reason of its usage and popularity. The survey showed that CI is widely adopted in popular projects and helped to decrease the time between releases.

Leppanen et al. [25] presented more frequent releases as a perceived advantage of CI, by studying the state-of-the-art CI practice in 15 Finnish software companies. They showed that projects using CI release twice as fast as projects that do not. Other researchers [26] also studied CI adoption by interviewing 27 developers at Ericsson's R&D to understand their perception of CI. These two works study CI usage and perception in industry, showing how CI gains popularity and is growing.

2.3 Build Systems

Build systems are the infrastructures converting a set of artifacts into an executable format. A build system has a critical role in software development. A poorly implemented build system frustrate developers and waste time [27].

Developers run the build process to compile and test the changes they made in the code. Then, they must wait for the build process while it is executing. Build tools like make, ANT, and Maven examine the last modification time of output and input files to perform only the commands required to update an executable. The waiting period frustrates developers and impacts on their productivity [5]. Substantial research has been performed to accelerate the build process. Adams et al. [28] proposed to recompile by semantically analyzing the changes performed in a file to check if it and any of its dependencies must be recompiled. Yu et al. [29] improve build speed by removing unneeded dependencies among files and unnecessary code from header files. These two approaches accelerate incremental compilation, which is not performed for CI, it always builds from scratch.

Suvorov et al. [30] examined successfully and failed migration to a different build system in two open-source projects, Linux and KDE, and outline four major challenges faced by the migration from one build system to another due to missing features of the build system. If the maintenance effort associated with the build system grows, developers choose to migrate to another build system. They stated that failed migrations usually do not collect enough build requirements prior to prototyping the migration.

Adams et al. [6, 31] analyzed the evolution of the Linux kernel build system. They studied the modifications in the size of the kernel's makefiles (SLOC), and dependencies in different releases. They found that build system expanded and must regularly be maintained [32]. They found primary evidence of improving complexity in the Linux kernel build dependency graphs. They showed that the build complexity co-evolves with the program source code at

the release level.

McIntosh et al. [7] empirically studied ten open-source projects and observed that between 4% to 27% of tasks involving source code modifications need a modification in the associated build code as well. They also mentioned that build code frequently evolves and is expected to include bugs due to high churn rate. Mcintosh et al. [33] studied the evolution of ANT build systems in four open-source Java projects, and stated that the build maintenance is mainly due to the code changes creation. They observed that the complexity of the build code and also the behaviour of the build system both increase and evolve.

2.4 Build Failures

While Schermann et al. [34] presented architectural issues as one of the major obstacles to CI adoption and build failures, we show configuration problems and platform dependent failures as one of the problems.

Kerzazi et al. [35] analysed 3,214 builds in a company during 6 months to study build failures. They found that the 17.9% of build that fail cost about 2,035 man-hours, if it takes one hour for each build to succeed. We studied 30 million builds and analyzed build results to avoid unnecessary builds and decrease these failure costs.

Rausch et al. [36] reported build failures in 14 open-source Java projects and categorized failures into 14 different groups. They performed their analysis in commit-level, and showed that more than 80% of failures are due to failed test cases, while we performed our analysis in release-level, and showed more than 80% of failures are due to programming, dependency, OS, and configuration issues. We investigated build failures in two groups, i.e., failures only affecting a few (minority) of the operating systems versus most (majority) of the operating systems, we reported 4.2% of majority failing builds categorized as test failures, in comparison with 10% failing tests in minority failures.

Kerzazi et al. [37] conducted a study of releases showing unexpected system behavior after deployment. Their findings show that source code is not the major reason of build failure, but defective configurations or database scripts are the main issues. Although our results verify that source code is not always the main reason of failure, we also concentrate on the importance of other factors (OS/environment) causing builds to fail.

Denny et al. [38] studied compile errors of Java code and observed that syntax is a crucial barrier for novice developers, and students submitted non-compiling code even in simple assignments. They claimed that 48% of build failures occur because of compilation errors. Another research on predicting performance in a programming course, Dyke [39] assessed the frequency of compile errors by tracking Eclipse IDE usage with novice developers, and checked for correlation with productivity. They showed that completed runs and successful compilation are related to productivity. We also study build failures and the reason of their occurrence, and categorize them into different groups.

Miller et al. [40] studied 66 build failures of Microsoft projects in 2007 and used CI as a quality control mechanism in one distributed, collaborative team across 100 days. They categorized these failures into compilation, unit testing, static analysis, and server failures. We investigated 791 build failures across 6 months, and describe a classification of build failures in a different domain, i.e., Perl and release level CI vs. Java and commit-level CI.

Vassallo et al. [41] classified CI build errors in 418 Java-based projects at ING, and 349 Java-based open-source projects hosted on GitHub that use Travis CI. The open-source and ING projects reported different build failure. This classification does not provide any information regarding the role of OSes and environment and how OS and environment can break the builds. In contrary, our classification explicitly considers OS and environment to understand what type of errors is the reason of build failure, we study build failures considering these two factors. We also determine to what degree OS and environment can lead to build inflation.

Mcintosh et al. [42] gathered the information of build source/test co-change in four open-source projects: Eclipse-core, Jazz, Lucene, and Mozilla. They derived some metrics like the numbers of files added/removed/modified, and provided a re-sampling approach to predict the build co-changes. We integrate these research results with our explanatory model of build failures of CPAN.

Finally, our study finds that API dependencies, such as missing libraries or the wrong API version cause, over 41% of builds to break. CPAN is an ecosystem of modules (APIs/libraries) and if build commands do not find any of the dependent modules, then the build will fail.

The API of an ecosystem is indeed a major element of development costs [43]. Zibran et al. analyzed 1,513 bug reports of Eclipse, GNOME, MySQL, Python, and Android projects, and among them, 562 bug-reports were about API issues. They found that about 175 of those issues were about API correctness.

Many researchers studied API for different objectives such as recommending appropriate APIs to developers when upgrading the dependencies of one module to a newer version [44]. McDonnel et al. [45] studied the API migration in the context of services. A previous work summarized this work [46], and here we summarize two works related to API.

Wu et al. [46] analysed 22 releases of the Apache and Eclipse frameworks and their client systems. They showed that the different API modifications in the frameworks affect the clients, then they categorized them into API modifications and API usages. To determine API usages and reduce the impact of API modifications, it is recommended to apply analyses and tools on frameworks and their client programs.

Tufano et al. [47] show that it is not feasible to revise most of the projects due to the missing dependencies for the older version of projects. Kula et al. [48] empirically studied library migration on 4,600 GitHub software projects and 2,700 library dependencies. They showed that although most of these projects rely on dependencies, 81.5% of the studied projects keep using their outdated dependencies. Our findings show more distribution of failures including dependency despite of accurate technical details.

Dig et al. [49] also conducted an analysis on five open-source systems (Eclipse, Log4J, Struts, Mortgage, and JHotDraw) to understand the API changes. They showed that 80% of changes are due to refactoring. However, as our study is in a higher level of granularity, we observed build failures related to programming issues where the majority of builds fail were 36.1% and 15.5% where minority of builds fail.

CHAPTER 3 RESEARCH PROCESS AND ORGANIZATION OF THE THESIS

This chapter presents the methodology and the structure of my thesis. This thesis wants to help practitioners understand the impact of operating system and environment on build results, and help researchers to consciously analyze build results. In Chapter 4, we analyze build failures in the CPAN build environment to find out whether prior findings on build failures and error types in Java [5, 41] also hold for Perl programming language and for release-level CI.

An earlier version of this work was published at the 14th International Conference on Mining Software Repositories [50]. Chapter 4 extends this work by adding three additional research questions involving a qualitative analysis of build failures. This extension has been submitted to the Springer journal on Empirical Software Engineering.

We aim to understand what factors play a role in the build process and in CI. This can help developers how to better contribute while integrating their code, as well as help researchers to identify the valuable OSes/environments to build on per day. Optimizing build execution will decrease the cost and effort required for performing builds, especially in large scale industries.

CHAPTER 4 ARTICLE 1: DO NOT TRUST BUILD RESULTS AT FACE VALUE –AN EMPIRICAL STUDY OF 30 MILLION CPAN BUILDS

Submitted at: Springer journal on Empirical Software Engineering (Extension of published paper at: the 14th International Conference on Mining Software Repositories)

Mahdis Zolfagharinia, Bram Adams, Yann-Gaël Guéhéneuc

Abstract

Continuous Integration (CI) is a cornerstone of modern quality assurance. It provides on-demand builds (compilation and tests) of code changes or software releases. Despite the myriad of CI tools and frameworks, the basic activity of interpreting build results is not straightforward, due to the phenomenon of build inflation: one code change typically is built on dozens of different runtime environments, operating systems (OSes), and hardware architectures. While build failures due to configuration faults might require checking all possible combinations of environments, operating systems, and architectures, failures due to programming faults will be flagged on every such combination, artificially inflating the number of build failures (e.g., 20 failures reported due to the same, unique fault). As previous work ignored this inflation, this paper reports on a large-scale empirical study of the impact of OSes and runtime environments on build failures on 30 million builds of the CPAN ecosystem. We observe the evolution of build failures over time and investigate the impact of OSes and environments on build failures. We show that Perl distributions may fail differently on different OSes and environments and, thus, that the results of CI require careful filtering and selection to identify reliable failure data. Manual analysis of 791 build failures shows that dependency faults (missing modules) and programming faults (undefined values) are the main reasons of build failures, with dependency faults typically being responsible for build failures on only few of the build servers. With this understanding, developers and researchers should take care in interpreting build results, while dashboard builders should improve the reporting of build failures.

Keywords: Continuous Integration, Build Failure, Perl, and CPAN

4.1 Introduction

Continuous integration (CI) is an important tool in the quality-assurance tool-box of software companies and organizations. It enables the swift detection of faults and other software-quality issues. A CI system, such as Jenkins, performs builds multiple times a day, for either

each patch currently under review, each new commit entering the version control system, or at given times of the day (e.g., nightly builds). It combines build and test scripts to run compilers and other tools in the right order, then test the compiled system [31, 5]. It notifies developers as soon as possible of build failures [14, 15]. A more coarse-grained form of CI is used by package repositories of open-source Linux distributions (e.g., Debian or Ubuntu) and of library repositories, like Maven or CPAN. Their CI systems only receive official releases (instead of every commit), yet those must be built and tested before publishing the new releases. Thus, a CI system may perform dozens or hundreds of builds a day.

Contrary to popular belief, a single commit or release is not built just once, but separate builds are performed for different environments and operating systems (OSes), such as different Java versions or different versions of Windows. These multiple builds, for multiple environments and OSes, lead to the problem of build inflation: there are many build results across OSes/environments, which are not necessarily uniform, i.e., they do not fail or succeed for the same reasons for all environments and–or OSes. Thus, additional builds may bring diminishing returns. Testing in some environment/OS, like Darwin in Figure 4.1, would be more valuable than testing on other OSes, like FreeBSD, because Darwin builds have targeted fewer Perl versions—half of which succeeded—while the vast majority of FreeBSD has been failing. Future FreeBSD builds are hence expected to fail, while the Darwin builds could either fail or succeed.

The software-engineering research community, although interested in CI, e.g., the MSR'17 mining challenge was about CI [1], lacks knowledge on the breadth and depth of the adoption of CI by software companies and organizations. We need answers to questions such as does CI's advantages surpass its disadvantages? Do developers use CI, and what are its limitations? How does CI help developers? Consequently, we study the impact of build inflation, in particular its resulting bias, on build failures and CI. We empirically study 30 million builds of the Comprehensive Perl Archive Network (CPAN) [13] performed between 2011 and 2016 and extracted from its CI environment. We cover more than 12,000 CPAN packages—called distributions in the context of CPAN and in the following, 27 OSes, and 103 environments. We answer the following seven questions:

— RQ1: How do build failures evolve across time?
— RQ2: How do build failures spread across OSes/environments?
— RQ3: To what extent do environments impact build failures?
— RQ4: To what extent do OSes impact build failures?
— RQ5: What are the different types of build failures?

1. `http://2017.msrconf.org/#/challenge`

— RQ6: To what extent do OSes impact build failure types?
— RQ7: To what extent do environments impact build failure types?

We show an inflation in the numbers of builds and build failures, which hide the reality of builds in noise, e.g., comparing the results of millions of builds on Linux with thousands of builds on Cygwin can lead to wrong conclusions about the quality of Perl releases on Cygwin in comparison to Linux. We also show that unnecessary builds, e.g., on OSes/environments, for which we already have many builds, could be avoided by investigating the impact of environments and OSes on build failures. Our observations provide empirical evidence of the bias introduced by build inflation and provide insights on how to deal with this inflation. We provide the largest quantitative observational study to date on build failures and build inflation. The results of our study form the basis of future qualitative and *quantitative* studies on builds and CI. They can help researchers to better understand build results. They can also help practitioners to prioritize the environments/OSes with which to perform continuous integration, in terms of expected returns, i.e., numbers of builds vs. chance of failures.

We extend our previous work [50] by analyzing the types of build failures and the impacts of different environments and operating systems on failures and their types. We thus can distinguish between OS dependent and independent build failures. We then categorize these build failures into 13 groups based on the reasons of the failures. We added RQ5, RQ6, and RQ7 to describe each of these categories and impacts. Finally, Section 4.5 also provides a detailed comparison of the obtained build failure types with those reported in the literature.

We organize our paper as follows: Section 4.2 presents background information on the CPAN CI environment and major related work. Section 4.3 describes our observational study design while Section 4.4 presents our observations, followed by their discussions in Section 4.5. Section 4.6 describes threats to the validity of our observations and discussions. Finally, Section 4.7 concludes with insights and future work.

4.2 Background

4.2.1 CPAN

Overview: This section provides an overview of the software ecosystem whose build results we are studying in this paper, i.e., the Comprehensive Perl Archive Network (CPAN). Similar to Maven and npm for Java and Node.js, CPAN is a ecosystem of *modules* (APIs/libraries) for the Perl programming language. It contains more than 255,000 Perl modules packaged into 39,000 *distributions*, i.e., packages that combine one or more modules with their documentation, tests, build and installation scripts. Each distribution can have one or more

versions. To simplify terminology, in the following, we refer to a distribution of a set of modules as a "dist" and to a distribution version as a "distversion".

Build Reports: CPAN implements its own continuous integration system, which builds and tests any new beta or official version of a dist on a variety of operating systems (e.g., Windows vs. Linux) and runtime environments (e.g., Perl version 5.8 vs. 5.19). If successful, the new distversion can be made available to CPAN users. Unlike Travis CI and Jenkins, but similar to Maven and npm, CPAN CI does not work at commit-level, but at release-level. CPAN depends on its own build servers and build servers owned and hosted by volunteering CPAN members. Thus, distversions are not guaranteed to be built and tested on every OS and environment version (cf. white cells in Figure 4.1).

As Perl is an interpreted language , build scripts typically process or transform code and data instead of mere compilation. The test scripts then allow to verify that the modules in the dist are working correctly on a given OS and environment. For each build (we will refer to the execution of build and test scripts for a given OS and environment as a "build"), CPAN generates a build report and all reports for a given version of a dist are summarized into an overview report, as shown in Figure 4.1. This Figure shows the summary of build results for version 0.004002 of the "List-Objects-Types" dist for environments 5.8.8 to 5.19.3 (left column) and OSes CygWin to Solaris (top row). Red cells indicate that all builds for a combination of OS and environment failed, green cells show that all were successful, red/green cells indicate that some builds failed, and orange cells represent unknown results (e.g., build or tests were interrupted).

Rest API: CPAN provides a RESTful API [51] and a Web interface [13] to allow complex queries on all publicly-available modules and dists. Furthermore, the whole history of CPAN and all of its modules and dists are accessible via the GitPAN project [2]. These data sources allow to conveniently access CPAN for analysis, providing access to build results and module meta-data, such as modules names, versions, dependencies, and other helpful information:

— GUID: a unique global identifier that identifies each dist.
— CSSPATCH: a value (`pat`) or (`unp`) indicating if this module was tested with a patched version of Perl.
— CSSPERL: a value (`rel` or `dev`) indicating if this module was tested with a release or development version of Perl.

2. https://github.com/gitpan

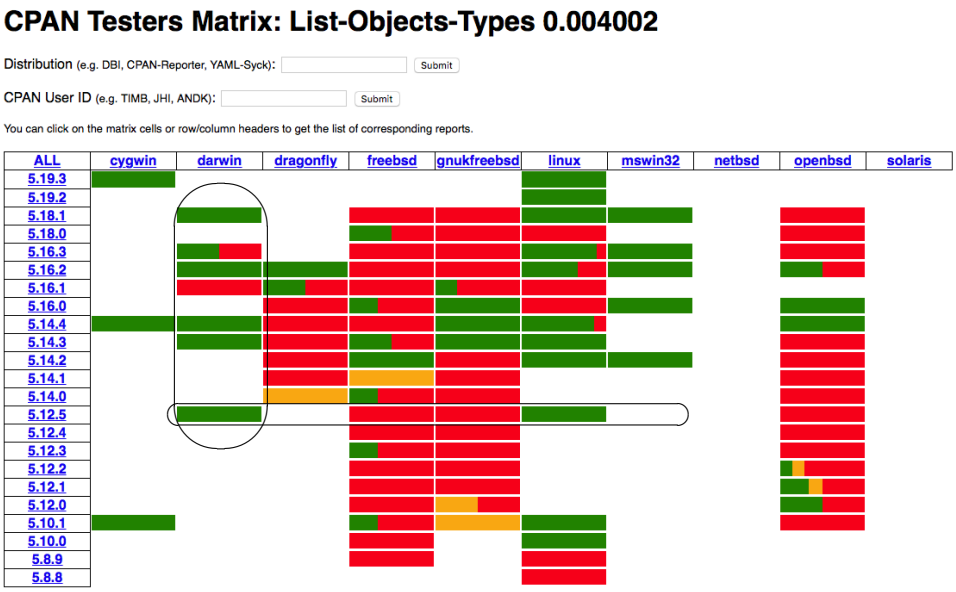

Figure 4.1 Example of CPAN build report summary. A vertical ellipse represents an "environment build vector" (RQ3), while a horizontal ellipse represents an "OS build vector" (RQ4).

4.2.2 Related Work

There exists previous work related to build, build failures, and CPAN or other ecosystems. We summarize now the studies most relevant to our own study.

Denny et al. [38] investigated compile errors in short pieces of Java code and how students fixed these errors. They showed that 48% of the builds failed due to compilation errors. Similarly, Dyke [39] assessed the frequency of compile errors by tracking Eclipse IDE usage with novice programmers.

Suvorov et al. [30] studied two popular open-source projects, Linux and KDE, and found that the migration from one build system to another might fail due to missing features. Adams et al. [6, 31] analyzed the changes to the Linux kernel build system and reported that the build system grew and had to frequently be maintained. McIntosh et al. [33] replicated the same study on the ANT build system. We complement these studies with our explanatory model of build failures for the build system of CPAN.

Other CI systems, similar to CPAN, but at the granularity of commits instead of releases, are Jenkins, Hudson, Bamboo, and TeamCity [19]. Stahle and Bosch [2] surveyed CI practices and build failures in such CI systems. They observed that test failures during builds are sometimes accepted by developers because developers know that these particular failures will be fixed later [52]. We complement this study by showing the impact of OSes and

environments on build failures within CPAN particular CI system.

Seo et al. [5] reported on a case study performed at Google, in which they assessed 26.6 million builds in C and Java. They showed that there exist multiple patterns of build failures and focused on compilation problems to conclude that dependencies are the main source of compilation errors, that the time to fix these errors varies widely, and that developers need dedicated tool support. We also showed that there exist multiple patterns of build failures. We also took into account the impact of OSes and environments on failures.

Vasilescu et al. [24] conducted an empirical study about CI usage on 246 projects from GitHub. They showed that CI significantly improves the productivity of GitHub teams. However, despite providing evidence on the benefits of CI, they do not provide any detailed information about CI usage, such as the consequences of continuously integrating many builds. Our findings provide evidence on the occurrence of inflation in builds, as CI has become more popular over time.

Hilton et al. [3] assessed 34,544 open-source projects from GitHub, 40% of which use CI. They analyzed approximately 1.5 million builds from Travis CI to understand how and why developers use CI. They found evidence that CI can help projects to release regularly. Leppanen et al. [25] also reported more frequent releases as a perceived benefit of CI by interviewing developers from 15 companies. Two other works [40, 26] have performed case studies on the use of CI and found a positive impact of CI. This growing usage of CI makes it important to perform a larger study of CI and builds.

Our study shows that 39.4% of the builds failing on only a few of the OSes are due to API dependencies (missing modules and libraries) compared to 27.8% of the builds failing on the majority of OSes. APIs are indeed an important factor of development costs [43]. Previous work studied APIs for various purposes: (1) to recommend relevant APIs to developers during development and–or during changes, typically when upgrading the dependencies of one module to newer versions of other modules [44], and (2) to migrate APIs, in particular in the context of services [45]. We refer the interested reader to a previous article summarizing this work [46]. We summarize here only two relevant works related to APIs and builds.

Wu et al. [46] analyzed changes in 22 releases of the Apache and Eclipse frameworks and their client programs. They observed the kind of API changes in the frameworks impacting the clients and classified API changes and API usages. They suggested analyses and tools apply to frameworks and their client programs to identify different kinds of API usage and reduce the impact of API changes. We provide evidence that such API changes are the most important cause of build failures, with a different impact depending on the OSes and environments on which the builds are performed.

4.3 Observational Study Design

We now describe the design of our observational study. For the sake of locality, we present the research questions, their motivations, and their results in the next section.

4.3.1 Study Object

The object of our study is the impact of OSes and environments on build results to analyze the phenomenon of "build inflation", where an excessive number of builds on heterogeneous combinations of OSes and environments can introduce bias in build results. Such bias makes it difficult to interpret build results (and hence detect bugs), and should be taken into consideration by practitioners and researchers analyzing build results. In certain cases, one might even consider to elide (combinations of) OSes and environments from the CI server if those do not contribute useful information.

4.3.2 Study Subject

We choose CPAN to study the impact of OSes and environments on build failures because CPAN [13] provides the results of the automated builds of all Perl dists and distversions on dozens of OSes and environments. Hence, it provides a large and rich data set. Moreover, CPAN has a long history, even though it provides build data only at the release level.

Using the data sources mentioned in Section 4.2.1, we mined the build logs and meta-data of all distversions. Build logs contain the results of all CPAN builds, including the commands executed, build results (failed or succeeded), and the error messages generated by the build and–or test scripts. The META.yml meta-data files contain a dist name, version, dependencies, author and other dist-related information (e.g., supported OS/Perl version). Using the dist build logs and meta-data as the main data source for our observational study, we obtained a data set of 16 years of build results for 39,000 dists, 27 OSes and 103 (Perl) environments.

4.3.3 Quantitative Study Sample

First, we fetched the complete CPANtesters build repository, yielding 68.9 million builds over a period of 16 years, between January 2000 and August 2016. We found that most builds were performed between 2011 and 2016. Although for each OS, builds were performed on different OS versions and architectures, 10 OSes and 13 Perl environments stood out. In particular, each of these 10 OSes had more than one million builds, while each of these 13

environments had more than 800,000 builds. To reduce time and (to some degree) simplify our analysis, we filtered out the other OSes and Perl environments, reducing the initial data set to 62.8 million builds on 10 OSes and 13 environments (Perl 5.8 to 5.21, excluding 5.09) for a period of about 5.5 years between January 2011 to June 2016.

As mentioned in Section 4.2, every cell in Figure 4.1 shows all the builds for a combination of OS and Perl environment. Red cells show that all builds failed, green cells represent that all were successful, and red/green cells indicate that some builds failed (due to different architectures and OS versions). Following the different directions of the research questions, in RQ1 and RQ2, we count all builds of every cell, in RQ3 and RQ4, we summarize the build output of each cell by considering only the most common build outcome, while in RQ5 to RQ7, we summarize each cell build outcome by considering only the most recent failure.

Although we investigate 13 Perl major versions, e.g., 5.8, the dataset includes 103 Perl minor versions, e.g., 5.8.8. Results of a major version include results of all minor versions. This data set included dists with only one build as well as dists with thousands of builds. For example, 13,522 dists have more than 1,000 builds while 967 dists have less than 3 builds. Unfortunately, not all builds have build results and we cannot draw any reliable conclusion for dists with too few builds or too few versions, while modules with too many builds or versions might not be representative either. Consequently, we filtered out builds without corresponding data, we determined lower and upper thresholds for the number of builds and number of versions, and filtered out modules below or above those thresholds, respectively, as explained in the following.

Figure 4.2 illustrates the distribution of the median number of builds and the number of versions across all CPAN dists in our full data set (the number of dists within each quadrant is shown as well). Black lines show the lower and upper thresholds that we determined for the median number of builds and number of versions for each dist. By looking at the data distribution, we filtered out CPAN dists with less than 10 build results and less than 5 versions. Then, to determine the upper thresholds for filtering outliers, we used the following formula [53] based on the inter-quartile range: $ut = (uq - lq) * 1.5 + uq$ where lq and uq are the 25^{th} and 75^{th} percentiles and ut is the upper threshold.

Of the nine quadrants shown in Figure 4.2, only the central one is used in our study. After removing 849,638 builds without data and filtering out the other quadrant data, we obtained a final data set of 12,584 CPAN dists with about 30 million builds. Including other quadrants would increase our data set size, but might introduce noise in the form of outliers. The resulting quantitative study sample is used in RQ1 to RQ7.

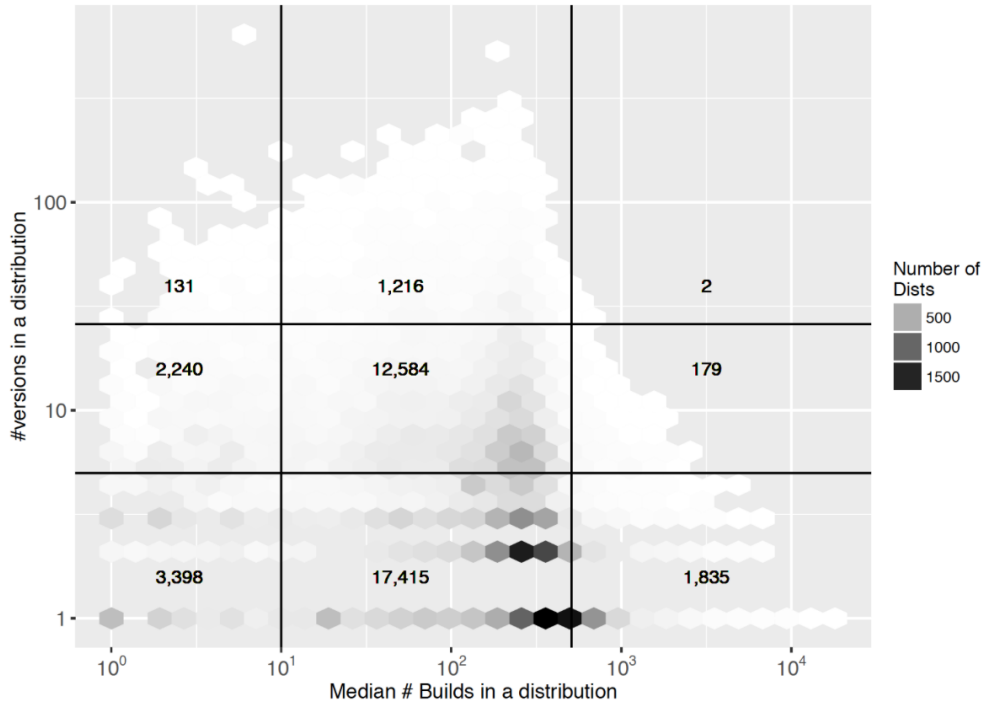

Figure 4.2 Distribution of the number of builds and versions across CPAN dists. This Hexbin plot summarizes where the majority of the data can be found (darker cells), where a cell represents the number of CPAN dists with a given median number of builds (x-axis) and the number of versions (y-axis). The black lines correspond to the thresholds used to filter the data, dividing the data into 9 quadrants. For each quadrant, the number of CPAN dists within it is mentioned on the plot. The central quadrant contains the final data set.

4.3.4 Qualitative Study Sample

We performed a qualitative study on build failures to categorize different types of failures that occur during the build process and to understand the impact of environment and OS on the types of failures that occur. We manually analyze CPAN build logs to identify the types of failures occurring in builds, then compare the frequencies of their occurrences.

First, we created a dataset of OS build vectors as it is shown in Figure 4.1 with at least one build failure among at least 10 OSes to compare build failures happening in only a minority of OSes (inconsistent failure) to those happening in most of the OSes (consistent failure), for a given Perl version (see RQ4, RQ6 and RQ7). Since for each combination of an OS/environment, multiple builds (and failures) can exist, with a maximum number of 1,362 failures and a median number of 64 failures per OS/env, we had to pick one build per vector element (OS/environment). In contrast to RQ3/4, we picked the most recent failure for each vector element, because we want to find out the most up-to-date reason of failures across all OSes/environments. This reduced the number of build failures from 76,748 to 1,421 across 804 OS vectors (note that most of the vectors did not contain any build failure, which is expected).

Furthermore, we observed that 4% of the selected build results are marked as unknown,while such unknown results were considered as failure in RQ3/4, we cannot do this for RQ5, RQ6 and RQ7 because we need to understand why a failure happened from its build log, so we ignored these unknown results. Note that although we removed 4% of the selected build results, none of the 804 OS vectors was removed entirely.

We distinguish the remaining OS vectors with 1 to 3 failing OSes into the group of "minority vectors" and those with 6 to 10 into the group of "majority vectors". We ignore vectors with 4 or 5 out of 10 failing OSes because those could have the characteristics of both minority and majority vectors. Thus, we obtained 752 vectors (963 failures) of minority failures and 52 vectors (458 failures) of majority failures.

Because 1,421 failures across 804 vectors is a large number, we randomly sampled 791 failures (confidence level of 95% and confidence interval of 5%) across 306 vectors, yielding a set of 52 majority vectors with 458 failures and 254 minority vectors with 333 failures, which we analyzed manually.

To categorize the different failure types, the 1^{st} and 2^{nd} author explored the build log of each analyzed failure and organized and labeled failures according to the fault responsible for the failure, using the "card sorting" technique [54]. Card sorting allows classifying the failures and the causes of these failures in 791 randomly-chosen build failures and label errors. This

technique is used in empirical-software engineering whenever qualitative investigations and classifications are required. Bacchelli et al. [55] used this technique to investigate code-review comments. Hemmati et al. [56] used it to examine survey analysis [57].

Card sorting involved multiple reviewing iterations. Initially, the first author investigated error messages in logs to analyze the reported symptoms of failures, then explored on-line reports and feedback for each error message to find evidence of and extract the real faults. Then, she added the error messages and their reasons (faults) into a card in Google Keep. Eventually, after analyzing the main rationale for each build failure and recording it onto a card, she grouped these cards into different categories based on a common rationale of build failures (faults). In the second and third iterations, the first and second author discussed differences among error types to categorize them. Major reasons for disagreement involved unclear error messages and too broad/narrow categories. Eventually, both authors agreed on six main categories of failures, containing nine subcategories. Figure 4.3 shows the categories that we studied in RQ5, RQ6, and RQ7.

4.4 Observational Study Results

We now present the motivations, approaches, and results of the seven observational research questions, RQ1 to RQ7.

RQ1: How do build failures evolve across time?

Motivation. This initial research question aims at understanding how often builds fail and whether the ratio of failing builds is a constant value or fluctuates across time. We investigate build inflation in terms of number of builds. Beller et al. [14] found a median of 2.9% of Java builds and a 12.7% of Ruby builds in Travis CI to be failing, while Seo et al. recorded failure ratios of 37.4% and 29.7% for C++ and Java builds at Google [5]. Unfortunately, apart from these lump numbers, not much more is known about the build failure ratios, in particular about the evolution of this ratio in time. Furthermore, all existing CI studies have targeted commit-level builds and tests, while CPAN is a package release-level build infrastructure, typical of software ecosystems.

Approach. To study the ratio of build failures in our data set, we consider all failing builds (red cells in Figure 4.1) and unknown build results (orange cells in Figure 4.1). When a particular OS/environment combination (one cell in Figure 4.1) saw multiple builds, we considered all of them in this RQ. For each CPAN dist in the data set of 30 million builds, we computed the ratios of build failures as $\#buildfailures/\#builds$. From 2010 to 2014,

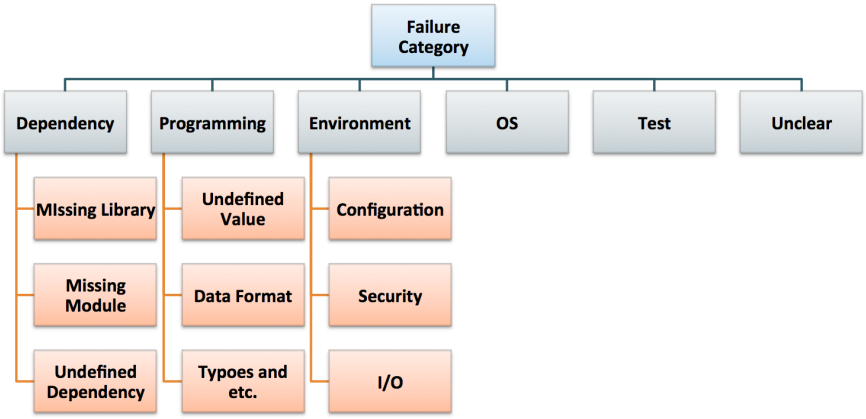

Figure 4.3 Hierarchy of fault types across all operating systems and environments.

each year, two Perl versions were released (release cycle of 6 months), so we investigated the evolution of failures per period of six months. We did not distinguish between OSes and environments in this RQ.

Findings. The median build failure ratio decreases across time from 17.7% in the first six months of 2011 to 6.3% in the first six months of 2016.

Figure 4.4 shows the distribution of failure ratios across all builds of all CPAN dists in the studied period of six years. From 2011 to 2013, the median failure ratio in the first half of a year is higher than that of the second half of the year, yet from 2014 on this trend is reversed. As the regression line in Figure 4.4 shows, the overall build failure ratio has a strong decreasing trend between 2011 and 2016, especially when taking into account the logarithmic scale used in the figure. This decreasing ratio might be due to several reasons, for example less builds being performed across time or less releases being made for dists. In the following, we explore these two hypotheses.

Until the first half of 2015, each year, more builds were being made, from one

Table 4.1 Number of builds, distversions, and average numbers of builds per distversion in each period of six months between January 2011 and June 2016.

	2011-A	2011-B	2012-A	2012-B	2013-A	2013-B	2014-A	2014-B	2015-A	2015-B	2016-A
Number of Builds	626	946K	1,860K	2,404K	3,021K	3,482K	3,625K	4,082K	4,827K	3,394K	2,891K
Number of Distversions	14	7,185	8,085	8,338	10,443	9,387	9,549	11,682	9,621	7,829	7003
#builds / #releases	44.7	131.7	230	288	289.2	371	379.6	349.5	501.7	433.5	412.9

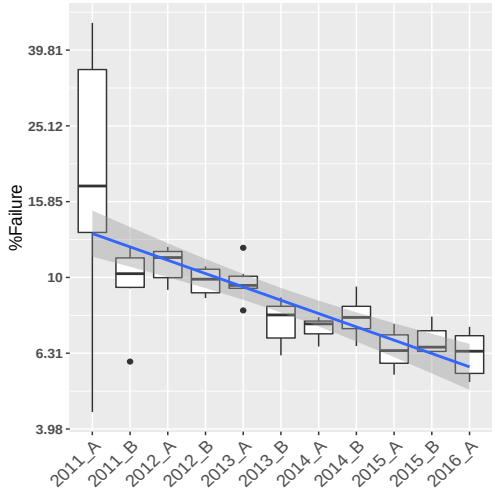

Figure 4.4 Distribution of failure ratios in six-month periods. The linear regression line shows the corresponding trend of the ratios over the six studied years.

(2011) to several million (2016). Table 4.1 shows the number of builds per period of six months. We observe that, even though ever more builds are executed, they seem more successful over time, i.e., there is an inverse correlation between numbers of builds and build failures. It is not clear why the second half of 2015 and first half of 2016 show a decreasing number of builds, however these observations might explain the plateau (instead of decrease) of median values for the rightmost box-plots in Figure 4.4.

The average numbers of builds per distversion shows a ten-fold increase from 44.7 to 412.9 across time, although there are some fluctuations from the second half of 2014 on (2014-B). To understand if the decreasing build failure ratio is due to a drop in the number of releases being made over time, we counted the number of releases of dists in each six month period. The average number of builds per release in Table 4.1 shows an increasing trend, growing from 44.7 in the first six months of 2011 to 501.7 in the first half of 2015 (with a slight dip at the end of 2014), after which the average ratio drops, but still remains higher than in 2014. We explain this observation as follows: although the numbers of builds dropped from the second half of 2015 on, the number of releases did not drop at the same rate.

Overall, the steady drop in build failure ratio can (at least partially) be explained by a strong increase in numbers of builds per release, i.e., strong increases in the numbers of builds and the numbers of releases per CPAN dist. While an increasing number of releases is typical for today's release engineering strategies [58], the increasing numbers of builds cannot be

explained intuitively. The next research question helps understand this build inflation by considering the impact of different OSes and environments on builds.

> *RQ1: The median build failure ratio decreases super-linearly across time, while the number of builds per distversion sees a 10-fold inflation.*

RQ2: How do build failures spread across OSes/environments?

Motivation. Build inflation seems to occur due to the needs for building and testing a release on different versions of the OS and Perl environment. Our hypothesis to explain the decrease of the build failure ratio observed in RQ1 is that a given new distversion is built multiple times in such a way that most of these builds succeed, while only a few fail. Although each OS and Perl environment, of course, can show deviating behavior (which is why multiple builds are performed in the first place), they are essentially building and testing the same features. Hence, feature-related faults are expected to trigger failures across all OSes and environments, inflating the numbers of build failures. Indeed, observing a feature-related build failure on one environment or OS theoretically suffices, additional builds are expected to fail as well. In contrast, OS- or environment-specific problems occur only for the problematic OS or environment, which is not trivial to predict. This deviation between different types of failures might not only explain our findings for RQ1, but also lead to bias in build results that must be addressed to avoid incorrect conclusions by build engineers and researchers.

Approach. For each build, we extracted build log data about the OSes and environments used during the build, then calculated build failure ratios per OS and environment. For the same reasons outlined in RQ1, we removed builds with unspecified status. Furthermore, as any CPAN community member can volunteer a machine for CPAN builds, a wide variety of hardware architectures and OS/environment versions are used. To make our analysis feasible, we again considered all build results recorded for a given operating system and environment. The analysis of the impact of hardware architecture is left for future work.

Findings. The analyzed CPAN distversions have a median of 179 builds, which took place on a median of 22 environments and 7 OSes. Figure 4.5 shows the distribution of the numbers of builds, OSes, and environments across all dists. While the numbers of OSes are more or less stable around 7, the number of Perl environments to test is much higher, while the total numbers of builds for a dist correlate with the product of both.

Through a manual analysis of the CPAN data, we observed that, when a new version of a dist

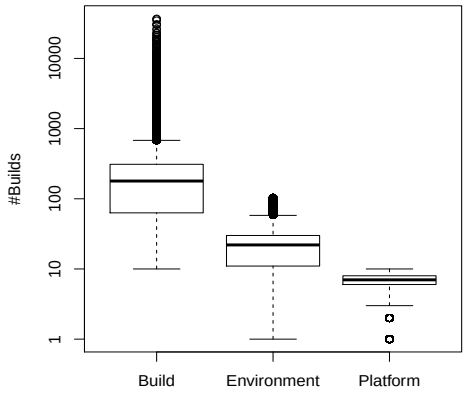

Figure 4.5 Distribution of the numbers of builds per CPAN dist, as well as of the numbers of OSes and environments on which these dists' builds took place.

is released, it should be built and tested on most of the supported OSes and environments to check if the new distversion is backward compatible with its application programming interface (API) [59]. Conversely, when a new OS or environment becomes available, most of the existing distversions that are not yet deprecated are rebuilt, which explains the inflation of the numbers of builds over time found in RQ1, but not yet the decrease of the build failure ratio.

Not every environment yields equally reliable build results. Figure 4.6 illustrates the evolution of build failures from Perl version 5.8 (released in 2002) until 5.21 (2015). Environments are shown on the x-axis, ordered by release date[60], while the y-axis shows the build failure ratios (blue; right axis) and the percentages out of all builds performed on a given Perl version (black; left axis). The jagged trend of the black line (percentages of builds) is surprising, suggesting that odd releases see substantially fewer builds than even ones.

Closer analysis showed that Perl uses a specific semantic versioning approach [60] where even release numbers, like 5.8 and 5.10, are official production releases (with maintenance releases, such as 5.12.1 and 5.12.2 mainly for bug fixes) and odd release numbers, like 5.11 and 5.13, are development releases. Hence, development releases are used less for builds and have less reliable build results. Although versions 5.19 and 5.21 were less failure-prone than their stable successors, other odd versions were more failure-prone. Furthermore, some builds have been performed in older environments, yet there was not enough build data to study these in details.

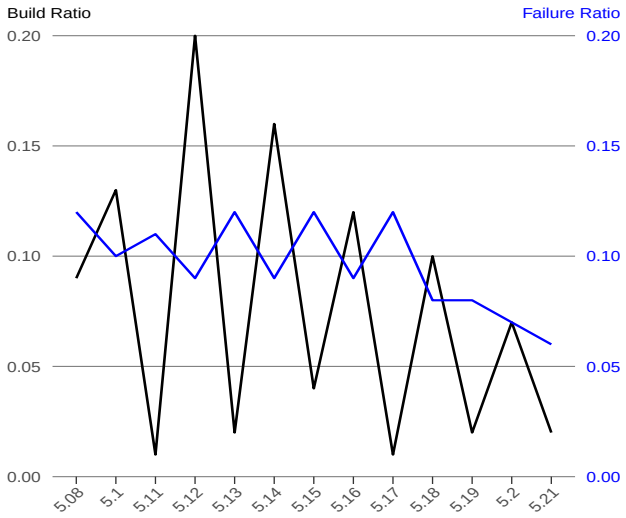

Figure 4.6 Distribution of the ratio of all builds performed on a given environment (black y-axis), and the proportion of those builds failing (blue y-axis).

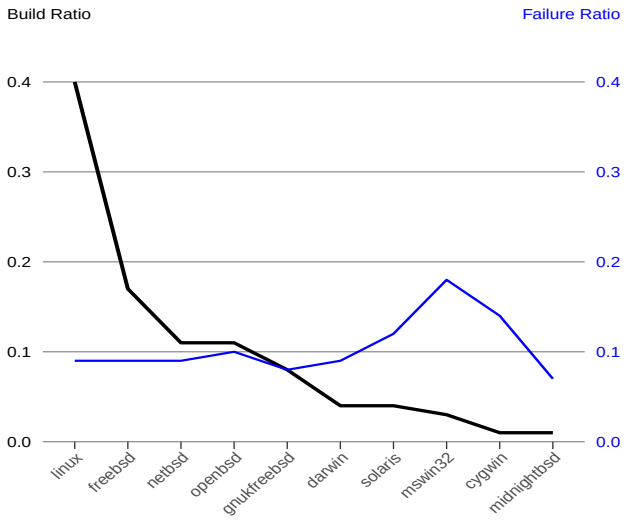

Figure 4.7 Distribution of the ratio of all builds performed on a given OS (black y-axis), and the proportion of those builds failing (blue y-axis).

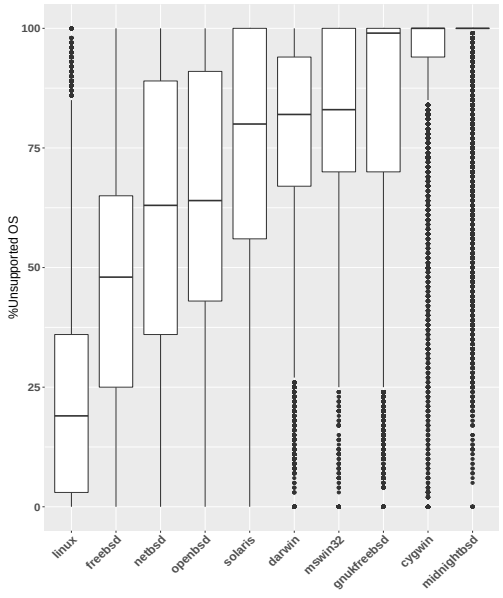

Figure 4.8 For a given OS, the distribution across CPAN distversions of the percentages of environments for which **no** builds have been performed.

The most failure-prone build OSes are Windows (%18), Cygwin (%14), and Solaris (%12). Figure 4.7 shows the percentages of builds and failing builds for different OSes. It shows a clear difference between BSDs/Linux on the one hand and Windows/Cygwin/Solaris on the other hand. The former cluster of OSes has a substantially larger numbers of builds than the latter cluster, while the percentage of failures is lower. Hence, similar to environments, the build results on some OSes are less reliable than others, because they might just indicate a lower popularity of those OSes in terms of builds and development. For some OSes like MidnightBSD or Cygwin, we have very few numbers of builds. Therefore, build results are based on small amount of data, which might not be as reliable as the results for Linux with huge numbers of builds. Finally, the average and median numbers of OSes on which each CPAN dist is built is 7. This number excludes OSes that were filtered out in Section 4.3.3 due to very low numbers of builds.

The fact that some OSes are more failure-prone and less popular amongst developers than others can also be observed when counting the numbers of times when no build is performed for a distversion on a given combination of Perl version and OS. These cases correspond to the empty cells in Figure 4.1. The resulting distribution of the percentages of missing builds across CPAN distversions for each OS is shown in Figure 4.8. We observe how the most incomplete OSes correspond to those with the most build failures in Figure 4.7. Therefore,

anyone interested in studying build results for CPAN should not blindly trust the build results for the least supported OSes, such as MidnightBSD, Cygwin, and Windows, because there are too few builds to be reliable and meaningful.

Due to lack of build resources, not all combinations of environments and OSes are built, hence the less important OSes have less build/test results, which lead to build inflation and makes interpreting build results harder. Spending more time and effort to run the valuable builds would help reducing inflation in builds.

> *RQ2: The OSes with the least builds have the most failures. Build engineers should not trust the build results blindly, especially for the least supported OSes, as their build failures are impacted by build inflation.*

RQ3: To what extent do environments impact build failures?

Motivation. In RQ2, we analyzed the impact of OSes and environments on the numbers of builds and the build failure ratio to better understand the increase in the number of builds. Although we found empirical evidence of build inflation due to repeated builds on different combinations of OSes and environments, we could not yet explain why the ratio of build failures has been *decreasing* over time. We suspect that build failures specific to one environment version will only register once, while builds of an abandoned distversion will fail on all environments and count multiple times. Similarly, OS-specific faults will register less build failures than faults in OS-independent code. Not all such failures are equally valuable to analyze. Therefore, this research questions and the next study the impact of the environment- (RQ3) and OS-specific (RQ4) build failures.

Lehman's 7^{th} law of software evolution states that "the quality of an E-type system will appear to be declining unless it is rigorously maintained and adapted to operational environment changes" [61]. As each CPAN distversion is immutable (any changes will generate a new distversion rather than updating an older one), once a distversion starts to fail on a given Perl version for a given OS, it is expected to keep on failing on future Perl versions on that OS, unless the Perl developers fix the Perl APIs. Hence, this RQ aims to understand how often environments break the builds and to what degree a failing build can recover again or is doomed to keep on failing.

Approach. For each distversion and OS, we consider the chronological sequence of build results across environments as binary vectors ("environment build vectors"), where a 0 marks

a failed build and a 1 a successful build. We ignored environment versions with missing builds: e.g., the environment build vector for Cygwin in Figure 4.1 is $[1, 1, 1]$ because we ignore white cells (missing builds). For a given OS and environment, some builds could be successful while others may fail (cells colored partially red and partially green in Figure 4.1), so we used the majority vote to summarize these build results into 0 or 1 values for use in the environment build vectors. If 50% or more of the builds for a given OS environment failed, the environment is said to be failure-prone (0 in the vector).

We then analyze the four possible patterns of build failure evolution and study these patterns across our environment build vectors to identify environments that are more failure-prone and tend to keep on failing. These four patterns are summarized in Table 4.2. For example, in Figure 4.1 OpenBSD's failure pattern is 0-0, because version 0.004002 of the List-Objects-Types dist started out and ended up failing on multiple environments, with some successful builds in the middle (Perl versions 5.14.4, 5.16.0, and 5.16.2). Similarly, the pattern for Cygwin is 1-1 (missing builds are ignored), while that for Linux is 0-1, with some fluctuations in the middle.

Findings. For 77% of the environment build vectors, builds succeed across all Perl versions. For 12%, the builds eventually succeed towards the most recent Perl versions. For the remaining 11%, the builds eventually fail or never succeeded at all across Perl versions. Figure 4.9 shows the percentages of environment build vectors matching each pattern in Linux. Blue bars show the percentage of "pure" matches, i.e., matches that do not include the optional part (between parentheses) of the patterns in Table 4.2. The blue bar for pattern 1-1 represents environments in which builds always succeed. Gray bars match the full patterns, including the optional fluctuations from 0 to 1 or 1 to 0, where the APIs of the Perl versions temporarily behaved differently than before.

Our observation provides evidence for Lehman's 7^{th} law of software evolution because, for

Table 4.2 Total percentage of occurrences of the four different build failure evolution patterns, across all OSes. "Pure" refers to occurrences of the patterns without fluctuation (e.g., $[1, 1, 1]$), while "Noisy" refers to occurrences with fluctuation (e.g., $[1, 0, 1]$).

Description	Pattern	Name	Pure	Noisy
Mostly Succeed	1+ (0+ 1+)*	1-1	77%	3%
Mostly Fail	0+ (1+ 0+)*	0-0	6%	1%
Eventually Fail	1+ (0+ 1+)* 0+	1-0	3%	1%
Eventually Succeed	0+ (1+ 0+)* 1+	0-1	8%	1%

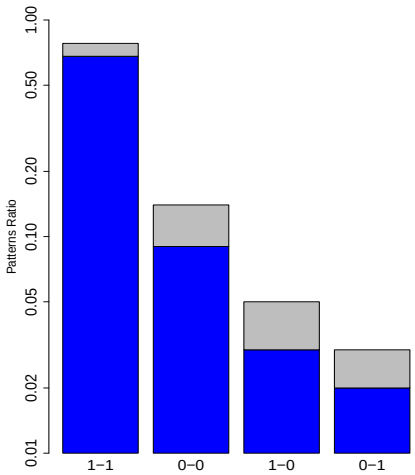

Figure 4.9 Percentages of the four build failure evolution patterns for the Linux OS. The blue bars represent occurrences of the pure patterns (no fluctuation) while gray bars are noisy occurrences (including fluctuations).

12% of the analyzed tuples (3% noisy for 1-1, 8% pure for 0-1, and 1% noisy for 0-1), changes to newer environments fixed previous build failures. These changes are changes in which an API is removed from a Perl version before being added again or where the implementation of an API changed behavior. For example, in dist version “Any-Template-ProcessDir 0.05”, Linux follows the 1-1 pattern: although it breaks in Perl version 5.13, it succeeds again from Perl version 5.14 onward.

However, 11% of the environment build vectors ended up failing in the most recent Perl versions and were unable to recover or never succeeded at all (0-0 and 1-0 patterns). To better understand these occurrences, we counted the numbers of trailing zeroes in the corresponding vectors as a measure of the time (in terms of the number of Perl versions) during which builds have been broken for a given OS. After normalizing by the total number of Perl versions on which builds were made, we found that 0-0 and 1-0 environment build vectors for Linux, Freebsd, and Openbsd have been broken for the shortest amount of time (trailing build failures account for 20% of the builds), while Cygwin environment build vectors have been broken the longest (50% of all builds). Windows, Darwin, Solaris, and Gnukfreebsd are in between those extremes (33% of builds).

As many builds are performed daily just to check backward compatibility and to verify code changes, we expect to have more and more consistent successes/failures. While each additional build could identify a failure (and hence contribute value), many builds on certain

OSes have a predictable outcome and could be avoided.

> *RQ3: For 77% and 6% of the environment vectors, builds consistently succeed or fail, respectively, across all environments. In other words, only for 17% of the vectors, build results can provide surprising information.*

RQ4: To what extent do OSes impact build failures?

Motivation. Although every new build on a new OS can bring new information, the amount of test results across environments and OSes is not homogeneously distributed, which gives some failures more weight than others. Except for a brief mention of different build environments in Travis CI [14], related work has not yet studied the impact of OSes on build results. Similar to the idea of build failures being specific to certain environments, this RQ analyzes the degree to which build failures are specific to certain OSes. If one OS is less popular than others, it might have seen less testing or some features might not have been ported over, causing test scripts to fail. Such failures would receive less weight in the build results than builds failing consistently across all OSes. Hence, we are interested in measuring whether certain OSes are indeed more failure-prone than others. Moreover, if build results of specific OSes are similar (BSDs), then we have the choice of testing only one of them for each distversion and environment.

Approach. To determine the consistency with which a build fails across all OSes, we build OS build vectors. Instead of summarizing build results across all environments for a given OS (environment build vectors), an OS build vector summarizes build results of a distversion across all OSes for a given Perl version. Again, a 0 value indicates build failure, while a 1 indicates build success.

In contrast to RQ3, in RQ4, we do not study chronological differences in build results, but rather how consistent builds fail across OSes. It is easier to have consistent build failures when a build is only done on three or four OSes rather than on ten, so we split our analysis across OS build vectors of different lengths, from three up to ten. Hence, we cluster the vectors into separate sets C_i, as follows:

$$\begin{bmatrix} B = \{\text{OS build vectors across all distversions}\} \\ C_i = \{b \in B \mid |b| = i\}, \forall i : 3 \leq i \leq 10 \end{bmatrix} \tag{4.1}$$

$$C_i \begin{cases} C'_i = \left\{ b \in C_i \mid \sum_{j=1}^{i} b_j = \text{i or } 0 \right\} \\ C''_i \begin{cases} C_i^M = \left\{ b \in C_i \mid 0 < \sum_{j=1}^{i} b_j \leq \frac{i}{2} \right\} \\ C_i^m = \left\{ b \in C_i \mid \frac{i}{2} < \sum_{j=1}^{i} b_j < i \right\} \end{cases} \end{cases} \tag{4.2}$$

For a given vector length i from three to ten, the set C_i is the union of the set of vectors that consistently failed or succeeded (C'_i), the set of vectors where a majority of OSes had failing builds (C_i^M), and the set of vectors where a minority of OSes had failing builds (C_i^m). The former two sets give a high weight to build failures, because most—if not all—of the OSes fail to build, while the latter set consists of build failure anomalies because few OSes have a different build outcome than the majority of the OSes. (We do not distinguish between sets of OSes (e.g., {CygWin, Windows, Linux} vs. {Darwin, Solaris, NetBSD}), only between lengths of vectors.)

Table 4.3 shows the percentages of vectors with inconsistent builds as well as how often those are caused by a minority of failing builds. If for a given vector, a minority of m OSes has a failing build, this counts as $\frac{1}{m}$ for each of the OSes. The more OSes fail together, the lower the weight we assign, because such build failures are less tied to one specific OS.

Findings. A median of 13.5% of OS build vectors fails inconsistently. Table 4.3 shows how the percentages of inconsistently-failing build vectors varies from nine ($N = 10$) to 16 ($N = 6$ or $N = 7$). Such build failures are specific to certain OSes. A median of 86.5% of build vectors either had no failed build or consistently failed on all OSes. The latter build failures are purely feature- or logic-related. This 86.5% of consistent build results (success/fail) show how numerous are builds, which leads to build inflation.

Out of the 13.5% inconsistent build vectors, a median of 71% have only a minority of failing OSes. Windows (30%), Linux (7%), and Solaris (7%) are amongst the OS minority that is failing. Windows is the source of most of the minority inconsistencies, which is likely due to its low popularity amongst CPAN developers as shown in RQ2. Linux causes more inconsistencies when being built with a small number of other OSes (small N), but is surpassed by Windows (and Cygwin) in larger sets of OSes (large N). Midnightbsd and Gnukfreebsd are the least inconsistent OSes, because they typically fail together with the other BSD OSes and Linux: they no longer belong to a minority but make up the majority of OSes.

Table 4.3 Percentages of vectors in C_i that fails inconsistently, as well as percentages of those vectors for which a minority of OSes is failing (C_i^m). The latter percentages are then broken down across all studied OSes (i.e., they sum up to the percentages in the third column).

N	$\%C_i''$ (out of C_i)	$\%C_i^m$ (out of C_i'')	Windows %	Linux %	Darwin %	Solaris %	Freebsd %	Openbsd %	Netbsd %	Cygwin %	Gnukfreebsd %	Midnightbsd %
3	10	61	13	15	3	4	11	6	5	2	2	0
4	12	50	13	11	3	5	7	4	4	1	2	0
5	14	67	22	11	6	6	7	5	5	2	2	1
6	16	65	27	8	6	6	5	4	4	3	1	1
7	16	75	33	6	8	8	4	4	4	5	2	1
8	14	81	38	4	9	8	2	4	4	7	3	2
9	13	90	44	3	10	8	1	3	3	13	3	2
10	9	94	50	2	6	8	0	1	2	21	2	2
Median	13.5	71	30	7	6	7	4.5	4	4	4	2	1

The fact that a median of 71% of the inconsistently failing build vectors has a minority of failing OSes, also indicates the presence of build inflation. Indeed, if one would know that a certain build failure is OS-specific (rather than feature-related), then having 1 or 2 successful builds on other OSes technically could suffice instead of having to run the build on all OSes.

Failure types that are known to occur consistently on most of the OSes, once identified on one OS for a particular distversion and environment, would not need further builds to be run on other OSes. Failure types with unclear occurrences across OSes need additional builds to circumscribe their scopes.

RQ4: Only a median of 13.5% of OS build vectors fails inconsistently, with a median of 71% of those having only a minority of OSes failing. This, and the fact that a median of 86.5% of build results fail/succeed consistently, again provides evidence of build inflation that complicates interpretation of the build results.

RQ5: What are the different types of build failures?

Motivation. Comprehending different build failures and their relationships with environments and–or OSes can help developers to solve future failures and help managers to prepare for future build failures. In particular, developers could use these insights to understand

better some of the build failures that they face and to have indications of the faults possibly causing these failures.

Approach. We used the qualitative analysis of section 4.3.3 to identify the different types of build failures. We manually analyzed the build logs and extracted the build failure information. Then, we categorized the failures according to the faults causing them.

Finding. We obtained six main categories of build failures, consisting of nine subcategories. We give a brief definition of each fault type and a sample of the resulting failures in each category. Table 4.4 summarizes the different categories obtained.

- The "Dependency" category is the first category of build failures, in which we have unfulfilled API dependencies, e.g., not installed module(s), missing libraries, and other missing files and dependencies. The reason for these failures is that some files do not exist on the CI server, while they do on the developers' machines. For example:
  - The fault: "Can't locate *.pm in @INC" occurs when there is a problem with installing modules in the runtime path (@INC) because a Perl module (i.e., a library implemented in Perl) cannot be found during the build: this fault belongs to the "missing module" subcategory. Failing tests because of a missing module: "Test::Perl::Critic required for testing PBP compliance" shows a test that requires the PBP module to check the presence of Perl Best Practices. Although the symptom of the build failure is a failing test, the reason for the failure here is a "missing module".
  - The fault: "Error while loading shared libraries: ?: cannot open shared object file" belongs to the "missing library" subcategory. This subcategory refers to OS-level library dependencies, such as unavailable C/C++ libraries, DLL conflicts in OSes, etc.
  - The "undefined dependency" subcategory represents failures like:"Makefile: recipe for target 'test_dynamic' failed.", which relates to the use of uninitialized dependencies.
- The "Programming" category relates to faults such as uninitialized values, implicit declaration of functions, typos, incorrect data types, and syntax errors. These faults imply that the code cannot be compiled or executed correctly, usually without any warning. For example:
  - A concrete example of undefined value is: "Use of uninitialized value $class_ip in concatenation(.) or string"
  - For typos: "prototype mismatch: 2 args passed, 3 expected"
  - For data type errors: "Non-ASCII character seen before =encoding. Assuming

Table 4.4 Failure types and their percentages in Perl versions with minority failures vs. majority failures

Fault Type	Subcategory	Description	Minority Failure	Majority Failure
			%	%
Dependency	Missing module	dist(s) not installed	35.8	27.8
	Missing library	Library not installed	3.6	0
	Unfulfilled dependencies	Other dependency issues	2	0
Programming	Undefined value	uninitialized value	7.9	26.4
	Typos, etc.	Source code issues	4	8.3
	Data format	Improper data type	3.6	1.4
OS		OS specific failures	12.9	12.5
Environment	Configuration	Configuration errors, e.g, wrong directories	12.6	6.9
	I/O	I/O errors,e.g. serial port issues	3.3	6.9
	Security	Configuration errors, e.g, permission denied	2	0
Test	Test	An automated test fail	6	2.8
	Test related to dependency	Failed test regarding to missing test module(s)	4	1.4
Unclear		Improper error message	2.6	5.6

UTF-8". These failures are related to faults in features or logic.

— The "OS" category covers faults that occur on specific operating systems, e.g., "your vendor has not defined POSIX macro VEOF" and "It seems localtime() does not honor$ENVTZ when set in the test script", which both happen on Windows. We observed that OS-related faults only occur in Windows, Cygwin, and Solaris, when the minority of builds fail. As we saw in RQ4, many build failures occur because of OS specific faults. Qualitative analysis of failures confirms that the lack of required implementation/configuration on different OSes can cause a build to fail, i.e., missing functionalities or different functionalities in certain OSes.

— The "Environment" category includes faults that occur when trying to access folders, resources, permissions, etc. For example:

  — If a build starts to run but fails during its process due to some configuration errors, then we categorize this failure as configuration, e.g., "MAKE failed: No such file or directory", display or serial port issues.

  — Permission issues, e.g., "Can't exec "vim": Permission denied" that we categorize it in security subcategory.

  — When a build cannot get I/O at the right time, it fails and we categorize this failure

as due to its environment: “Could not execute: open3: Resource temporarily unavailable”

— The “Test” category occurs when an automated test fails like: “release-pod-syntax.t these tests are for release candidate testing”. The test category refers to failures during execution with a warning message, but results are semantically incorrect.

— The “Unclear” category represent faults for which an improper message appears in error logs.

RQ5: Build failures belong to six main groups and nine subgroups: dependency, programming, environment, OS, test, and unclear failures are the main categories.

RQ6: To what extent do OSes impact build failure types?

Motivation: This question aims to identify the most common faults resulting into majority failures (i.e., across many different OSes), then compare those with the most common faults resulting into minority failures to identify the reasons for builds to fail on only a few of the OSes.

Approach: We used the minority and majority datasets that we obtained manually from RQ5 to determine the prevalence of each fault (sub)category. Since for majority failures most of the OSes fail for the same reason (fault), we then focused on the minority failures to understand for each OS the different fault types due to which it fails across the minority builds it participates in. We then compared the results across the different OSes, yielding information about OS-specific fault types.

Findings: The most common reason of build failures overall are dependency faults, followed by programming faults for majority failures and environment faults for minority failures. Table 4.4 (last columns) shows the breakdown of the percentage of occurrences of each (sub)category of faults within the minority and majority build failures, while Figure 4.10 gives a visual overview of these numbers. The dependency faults, especially “missing module”, dominate both majority and minority failures followed closely (for majority builds) by the “undefined value” subcategory of programming faults. The popularity of programming faults among majority faults is to be expected, since an “undefined value” or “typo” is unlikely to be fixed just by changing the underlying operating system.

Windows, CygWin, and (to some extent) NetBSD are minority failing OSes experiencing a wide variety of faults. Figure 4.11 represents the distributions of build

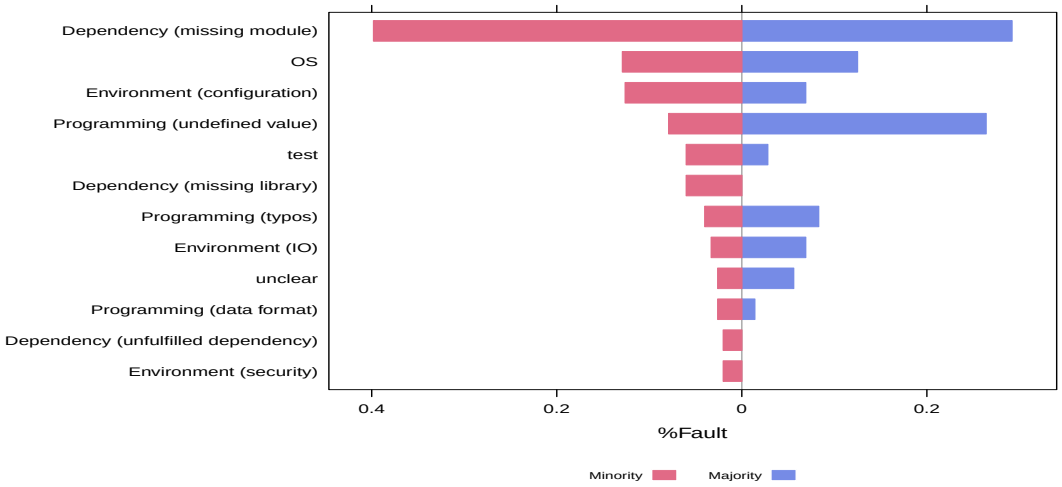

Figure 4.10 Distribution of fault categories in OS vectors with minority failures vs. majority failures

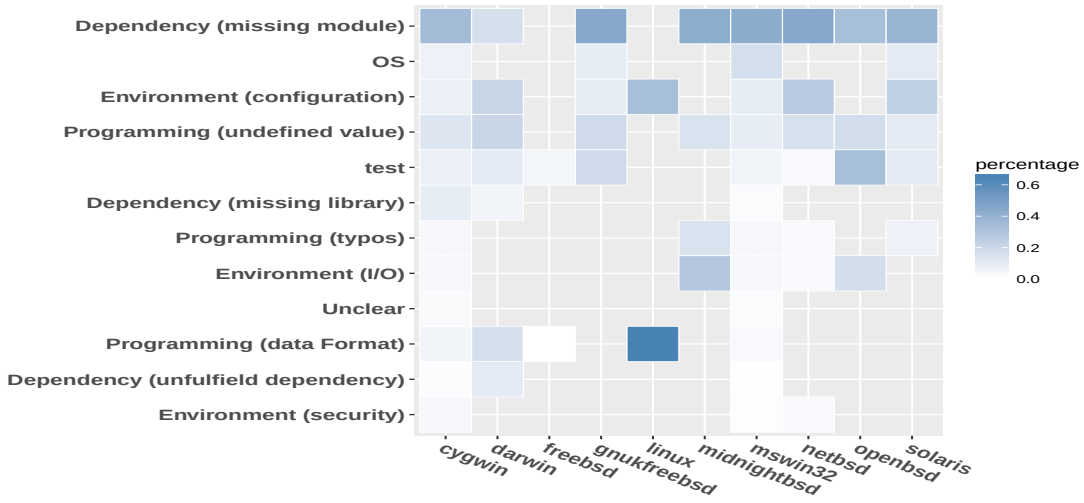

Figure 4.11 Distribution of fault categories across all OSes in the minority dataset.

fault types across all OSes. It shows that Windows and Cygwin were the most minority fault-prone OSes, confirming the results of RQ4. In addition, we now can see that this is because they experience a wide range of fault types when they have a minority failure. If we compare this to Linux, we can see how in 66.7% of the times when Linux is responsible for a minority failure, this is due to the "data format" subcategory of programming faults, while the remaining 33.3% this is due to the "configuration" subcategory of the environment faults.

RQ6: Missing modules (dependency fault) is the main reason of failures overall, with majority failures also commonly caused by programming faults and minority failures by environment faults. The OSes experiencing many minority failures do so because of a wide variety of faults categories.

RQ7: To what extent do environments impact build failure types?

Motivation. To better understand the role of environments in build failures, we studied failures in different environments to find out which failures are temporary and which ones are permanent. The term "temporary" refers to the fact that, for a given OS, a failure only shows up for some of the environments, but is fixed in the more recent environments. The term "permanent" refers to a failure that shows up in all environments. By comparing temporary/permanent fault types between minority and majority failures, we might find additional evidence of build inflation, since permanent failures again would be highly predictable and lead to redundant builds on newer environments.

Approach. To determine whether a fault type is temporary vs. permanent, we leverage the four patterns of RQ3. Starting from the 791 manually analyzed build failures of RQ5 (333 minority failures and 458 majority failures), we then used regular expressions to automatically match each failure's corresponding error message across all Perl versions for the OS and distversion for which the failure occurred. Basically, for each failure, we tried to find all its occurrences within one column of Figure 4.1.

For example, we searched the failure "Can't locate *.pm in @INC" of distversion "Acme-CPANAuthors-0.23" on Windows for Perl version 5.14.4 across all other Perl versions for which a Windows build was made. We then analyzed the obtained environment vectors to understand how the manually observed build failures evolved according to the patterns of

RQ3. As failures due to API dependencies are significantly more numerous than others (see RQ6), we aggregated all non-dependency failures into one group and compared this group to build failures due to dependency faults.

Findings. Dependency faults for majority failures are more difficult to resolve than for minority failures. Figures 4.12 and 4.13 compare the distributions of build failures for the four patterns of RQ3 between dependency and non-dependency fault categories, in majority and minority builds respectively. We noticed that while 72% of the dependency faults of majority failures follow the 0-0 pattern, only 51% follow this pattern for minority failures. In contrast, the number of 1-1 matches double from 14% to 28%. This means that the dependency faults responsible for majority failures are more permanent, and hence harder to resolve, than those responsible for minority failures. For example, by fixing a dependency on the environment or underlying OS, a minority failure on Windows may be resolved, while doing this for majority failures on Windows seems much harder.

Non-dependency faults do not experience different behaviour between minority and majority failures. While Figures 4.12 and 4.13 show a slight increase in 0-1 and 1-0 pattern occurrences of non-dependency faults between majority and minority failures, the order of magnitudes of 0-0 and 1-1 pattern occurrences are comparable to each other. This is not surprising, since we found in RQ6 that the programming category dominates the non-dependency faults and such faults are hard to fix without touching the source code. Since all our results in this RQ consider, for each failure, only the distversion in which the analyzed failure occurred, the source code of the dist did not change (only the environment and/or OS might have).

The proportions of 0-0 and 1-1 patterns across all dependency and non-dependency faults are more numerous than the other patterns. This shows how consistently failing or succeeding builds produce repetitive information, i.e., build inflation. On the other hand, the patterns 0-1 and 1-0 (i.e., inconsistent build results) comprise a smaller proportion of builds.

RQ7: Dependency faults are more temporary in nature across environments for minority failures, and hence are more likely to be fixed in builds on future environments, than non-dependency failures. However, non-dependency faults did not exhibit such a difference between minority and majority failures.

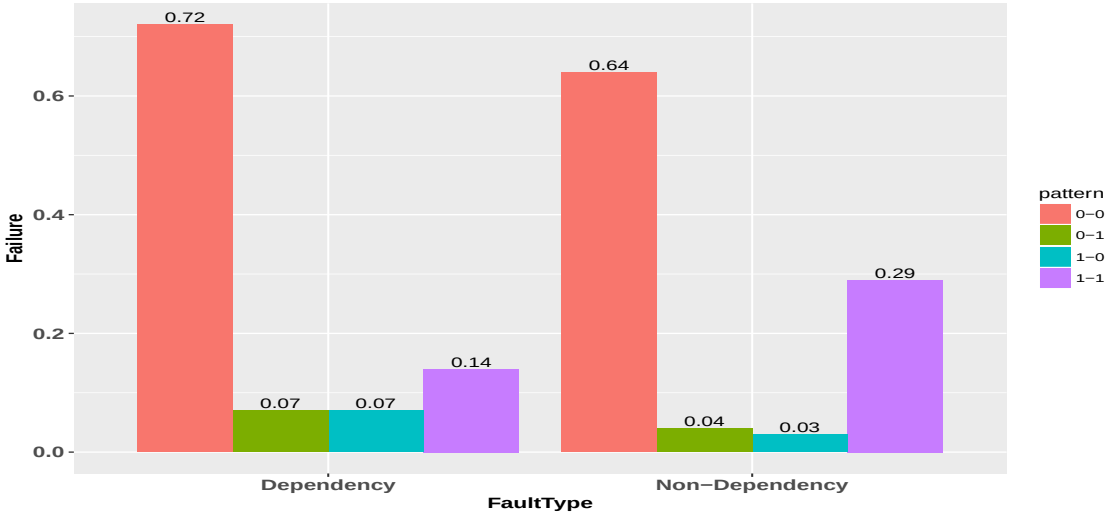

Figure 4.12 Distribution of failures due to dependency vs. non-dependency faults in different build patterns when majority (6-10) of builds fail. Y-axis shows the failure ratio and x-axis shows fault types and build patterns.

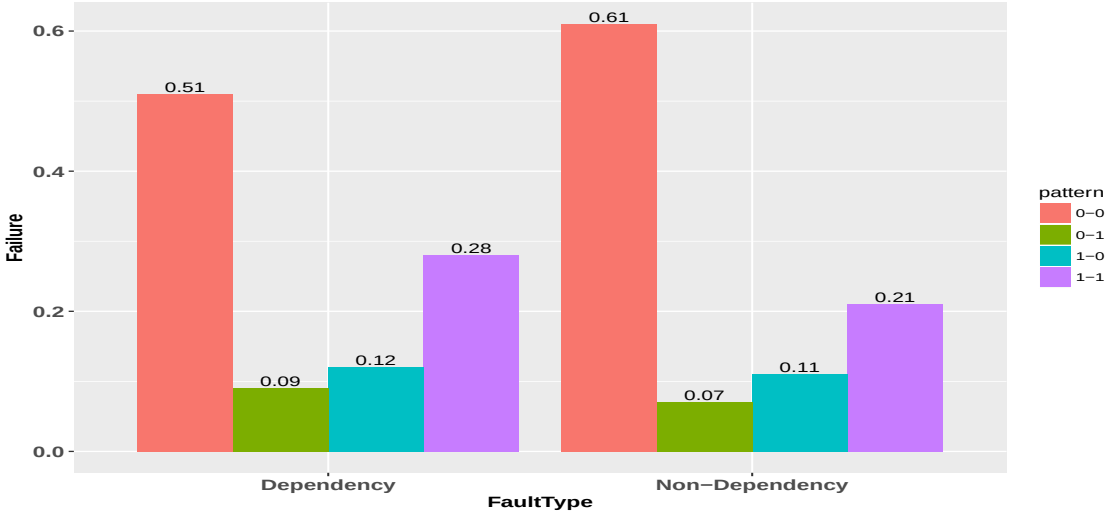

Figure 4.13 Distribution of failures due to dependency vs. non-dependency faults in different build patterns when minority (1-3) of builds fail. Y-axis shows the failure ratio and x-axis shows fault types and build patterns.

4.5 Discussion

After answering RQ1 to RQ7, we have a better understanding of the frequency of builds and build failures and the distributions of these failures across OSes and environments. We also have a better understanding of the faults leading to these failures. In particular, we understand better why, despite an inflation of builds across time, the percentage of build failures keeps on decreasing. We now validate our observations, in particular those about build OSes and environments and their impacts on build inflation, to explain the presence of build failures.

4.5.1 Explanatory Classification Model

We build an explanatory classification model with build failure as dependent variable and with OSes and environments as independent variables. We use random forest classifier for our models, which build an ensemble of decision trees that are used together to classify a given build [62]. We use 10-fold cross validation to evaluate the stability of the explanatory models, then calculate the area under the ROC curve (AUC) to understand how much better the models are than random guess (AUC>>0.5). We calculate what is the percentage of build failures classified correctly as build failures (true positive recall) and what is the percentage of successful builds classified as such (true negative recall). The higher these percentages, the better OSes and environments explain build failures.

We build explanatory models for different dists. Initially, we chose dists with more than 200 builds, having failure percentages between 20% and 80%. Thus, from 39,000 dists, we kept 12,584, out of which we kept 3,949 dists, so that we have enough builds to get a rational result from a random forest and to apply cross validation. To have enough data and avoid having thousands of small models, we then grouped related dists into groups based on the first part of their names, e.g., Acme, Net, Yahoo, etc. We thus built 677 explanatory models for 677 groups of dists.

The explanatory models are not useful in practice to predict build failures, because they only include OSes and environments, while ignoring other factors, notably factors related to source code and overall code quality. However, prediction is not the purpose of these models. We use these models to validate the extent to which knowledge of OSes and environments alone can explain build failures.

A median of 88% of successful builds and 80% of failing builds are correctly classified as such based only on information about OSes and environments. Furthermore, as shown in the bean-plots of Figure 4.14, most of the models have an AUC value

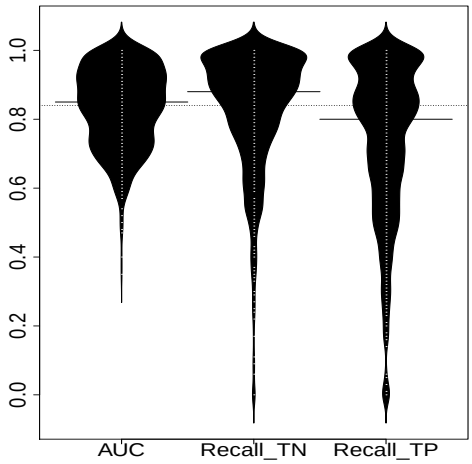

Figure 4.14 Bean-plot showing the distributions of AUC, true negative recall and true positive recall across all dists. The horizontal lines show median values, while the black shape shows the density of the distributions.

higher than 80%, which shows that the models clearly contain more knowledge than a random guess. By using the AUCRF algorithm [63], which implements a backward-removal process according to the primary ranking of the variables, we found that OSes have a higher explanatory power than environments for build failures, confirming our findings in RQ4.

4.5.2 Comparison to Prior Build Failure Research

This section compares prior research results with the build fault types identified in RQ5. While Schermann et al. [34] found architectural problems as one of the main obstacles for CI adoption and build failures, we found configuration issues and OS-specific faults as the major problems in general.

Rausch et al. [36], analyzed build failures in 14 open-source Java projects, found that more than 80% of the failures are due to failed test cases. Our study shows that more than 80% of the failures are related to programming, dependency, OSes, and configurations.

Beller et al. [14] conducted an extensive analysis of 1,359 projects in both Java and Ruby and observed that commit-level CI results are dependent on the programming language, i.e., Ruby projects tend to have ten times more tests and hence have a higher build failure ratio due to tests than Java. Compared to this commit-level study, our study focuses on release-level build results for one programming language.

Seo et al. [5] focused on compiler faults that happen in build processes in Java and C++, which are compiled languages while we studied Perl that is an interpreted language. However, similar to our findings, they observed that dependencies between components is the most common fault type.

Kerzazi et al. [35] conducted a study on 3,214 builds in a software company for six months to analyze build failures. They found that 17.9% of the build failures have a potential cost of about 2,035 man-hours, considering that each failure requires one hour of work to be fixed to succeed. They also studied [37] releases of a large e-commerce web app to show unexpected system behavior after deployment. They reported that source code is not the main cause of problems, but defective configurations or database scripts often lead to build failures. While our findings confirm their observations, we also showed the importance of other fault types related to OSes and environments.

Vassallo et al. [41] studied the distributions of CI build failures in 418 Java- based projects at a company and 349 Java-based open-source projects hosted on GitHub utilizing Travis CI. They observed how open-source and industrial projects present different distributions of build failures and categorized these failures into 15 categories. They did not consider OSes and environments, which can provide more information about how OSes and environments may lead to build failures. We tried to understand what type of faults cause build failures, considering the impact of OSes and environments on failures. We also tried to identify to what degree build inflation may be caused by OSes and environments.

Miller et al. [40] studied 66 build failures in Microsoft projects and categorized them into compilation (26%), unit testing (28%), static analysis (40%), and server (6%) failures. We studied 791 build failures and proposed another categorization of faults. We performed our analysis at release-level instead of commit-level.

Dig et al. [49] analyzed five popular open-source systems (Eclipse, Log4J, Struts, Mortgage and JHotDraw) to analyze API changes. They observed that 80% of the changes are related to refactorings.

Tufano et al. [47] showed that making changes is not feasible for many projects because of missing dependencies for earlier versions. Kula et al. [48] found that identifying transitive dependencies needs fixing many dependencies of a project. They showed many factors that should be considered when updating a library, including API migration consequences that should be addressed. We analyzed the CPAN ecosystem and found that API dependency is a major reason of build failures.

4.6 Threats To Validity

We performed this study on build data of CPAN (which is centered on the Perl language). We applied a methodology to categorized failures only in CPAN. Thus, we cannot generalize our observations and answers to other ecosystems and programming languages. Yet, the numbers of builds, OSes, and environments provide an interesting basis on which to perform future studies and against which to compare their results.

Regarding construct validity, we fetched the build information from the centralized CPAN build archive, which does not provide the full detailed reports on all errors occurring during builds. We filtered out some builds. Hence, even through we studied 68.9 millions builds and their results, it is possible that other failures/successes exist.

Furthermore, we observed that multiple builds may occur for a single combination of OSes and environments. These builds would typically exercise a distversion on different variants of an OS/environment and even on different hardware architectures. We aggregated all builds into one and ignored the architectures. We chose to describe a combination of OSes and environments as failing if at least 50% of its builds failed or had unknown results. This choice could impact our results, although they follow common sense. Future work includes analyzing in more details these operating systems and environments with multiple versions/architectures.

Regarding internal validity, while we studied the impact of OSes and environments on build failures, we did not perform in-depth analyses of the reasons for failures/successes of a given OS version, Perl version, or combination thereof, as well as other factors, like architectures, minor/major releases, developer experience, module complexity, etc.

From 39,000 distributions in CPAN, we chose those that were best fit for the analysis, similarly to recent work on Travis CI [14]. Results could be different with a different dataset.

4.7 Conclusion

Build results are a valuable data source for both researchers and practitioners. However, in this paper, we showed that we cannot trust these results at face value due to the phenomenon of build inflation, where the number of builds for individual code changes or package releases artificially is inflated to dozens of builds across different environments and OSes. The term "inflation" implies that not all those additional builds are equally useful, since it increases the importance of certain build failures, while it hides others. Whereas builds are supposed to give an indication of the quality of a product, we conclude that, taken at face value, build

results reflect more on portability across and problems with OSes and environments.

In particular, based on our study of 30 million CPAN builds between 2011 and 2016, and qualitative analysis of log files, we conclude that researchers and practitioners should be aware that:

— the number of builds for a given product can see up to 10-fold increases, while the build failure ratio seemingly decreases substantially (RQ1)
— a given product, like a CPAN distversion, is built on dozens of environments and OSes, many of which are not stable, popular or usually supported, and hence many repetitive builds are performed with predictable results, leading to build inflation (RQ2)
— the builds of a working product may fail due to changes in the environment, with only a small chance for recovery (RQ3)
— some OSes are notorious for failing the builds (RQ4)
— the most common build fault categories are dependency, programming, environment, OS and test faults (RQ5)
— dependency issues are temporary in environments with minority of failures, while in majority failures, non-dependency issues are temporary and are more likely to be resolved than dependency failures (RQ6)
— while missing modules is the main reason of failures in both minority and majority failures, majority ones are likely to occur due to programming faults (OS-independent), and minority failures due to configuration faults (OS-specific) (RQ7)

This work provides motivation for the software community to develop approaches to improve the developers' perception of CI, in the wake of build inflation, and its accompanying costs. Thus, we conclude that researchers interested in studying build results should analyze and select the results for the main environments and OSes, while ignoring other build results. Although this observational study is focusing only on CPANtesters, it is the larger to date observing build failures and build inflation across dozens of environments and OSes, and hence we hope it will serve as the basis of future qualitative and quantitative studies on builds and CI.

Acknowledgment

Part of this work was funded by the NSERC Discovery Grant program and the Canada Research Chair program.

CHAPTER 5 GENERAL DISCUSSION

In this thesis, we conducted an empirical study to identify the presence of build inflation. To analyze the impact of OSes and environments on build results, we looked at the frequency of build failures, the distributions of these failures across OSes and environments, and the faults leading to these failures. Therefore, we understand better why the build failure ratio keeps on decreasing.

In particular, we observed that inflation is present in:

— the needs for building/testing a release on variety of OSes/Perl environments, that basically build/test the same features. So, feature-related faults are supposed to trigger failures for all OSes/environments, therefore observing a feature-related build failure on one environment/OS theoretically is enough. In contrast, OS-/environment-specific issues happen just for the problematic OS/environment, which is not critical to predict. This deviation between different failure types leads to bias in the results that must be solved to avoid wrong interpretation by build engineers.

— build failures that specific to one environment version and should just register once, while builds of an abandoned distversion fail across all environments, so will count several times. likewise, OS-specific faults register fewer build failures than faults in OS-independent code.

— failure types that are expected to happen consistently on most of the OSes, so when they identified on one OS for a specific distversion and environment, would not required further builds to be run on other OSes.

— temporary/permanent fault types for minority/majority failures to find more evidence of build inflation, as permanent failures would be highly predictable and lead to repetitive builds on the most recent environments.

My thesis focused on the total number of builds that happened in release-level CI, not whether at least one build is triggered for a given release/commit. Therefore, every release/commit should have at least one build triggered for quality assurance activities to be performed. Yet, the main question of my work is “how many builds should be run per build trigger”? The various indications of build inflation in terms of repetitive/redundant builds across environments and OSes that we found mean that too many similar build failures for a given build trigger could over-emphasize a build problem or success, while at some point an additional build for a given trigger does not add much value anymore for developers.

Similarly, we found that certain faults are more temporary vs. permanent, which means

that many failures on the older environments will be resolved in the most recent ones. For example, dependency faults responsible for a minority of OSes failing are easier to resolve (i.e., are more temporary) than when they are responsible for most of the OSes failing.

Despite these two findings, which seem to suggest that (due to repetition) one could predict build failure and hence avoid to run certain builds, there is always a risk that an additional build might suddenly find a new fault. Therefore, although we can define some metrics to predict build failures, relying 100% on prediction is not necessarily safe. Hence, future work should consider both the costs (time until build failure is fixed again), risks (failure might slip through to the end user) and benefits (finding additional failures) of additional builds.

Finally, in our empirical study, we do not imply that certain OSes are more buggy than others. For example, the observations that the Windows build failures might be harder to interpret than the Linux ones, or seem more failure-prone, are mainly due to the inflation of Linux build results compared to non-Linux/BSD OSes. In other words, this thesis precisely warns for this kind of generalization of build failure results.

CHAPTER 6 CONCLUSION

We conclude this thesis with a summary of its contributions, some of its limitation, and possible directions for future work.

6.1 Summary

Build systems automate the conversion of the source code into executables, and are crucial for supporting changes. Build systems are beneficial for both researchers and practitioners, however due to the phenomenon of build inflation, according to which each code changes or new release produces several builds across different OSes and environments, we must take care when interpreting build results. This build inflation exaggerates the importance of particular failures, while it underestimates the importance of the others. To better understand the overhead enforced by building on different operating systems and runtime environments, we studied build results over time. We perform a large-scale empirical study in 30 million CPAN builds between 2011 and 2016 to validate our research hypothesis.

In Chapter 4, we analysed the evolution of build failure during 5.5 years, finding that build inflation happens at the release level due to an enormous number of builds on a variety of operating systems and environments that has diminishing returns. Our findings show:

- While the build failure ratio shows a sharp downward trend, the number of builds sees a 10-fold increase.
- Software projects are built on different environments and OSes, which are not equally reliable as they are not popular and-or commonly supported as build platform. A software project can fail due to changes in the environment, with a minor chance to succeed again.
- Similar to Seo et al. [5], we performed a manual analysis of build failures and observed that API dependency and programming issues are the major reasons of build failures.
- “Missing module" has the highest percentage in breaking the builds either in minority or in majority failures, while issues regarding the configuration of the machine are the 2nd highest reason of failure in minority failures.

6.2 Limitations and Future Work

I studied the impact of OSes and environments on build failures. I studied build failures quantitatively and qualitatively on ten operating systems and 103 runtime environments. My findings can be used for future studies on builds and CI. Future work should replicate my study at the commit level on other repositories like Travis CI. I could also analyse build results for different OS versions/architectures in more detail. In-depth analyses of the reason of failure of a given OS version/architecture, and developer experience also remain as future work.
The work presented in this thesis has some constraints:

- I only performed this study on the Perl programming language and on the CPAN ecosystem. Therefore, I cannot generalize our findings to other ecosystems and programming languages. However, I performed a large scale study on build results, on different OSes, and environments, which can provide an interesting basis for future studies.
- Although I studied 68.9 million builds, I filtered out builds without corresponding failure, so having a different dataset with different failures may lead to another results.
- Moreover, multiple builds may happen on every combination of OS and environment. Those builds would typically exercise a module even on different OS versions and hardware architectures. However, we ignored the OS version and architecture dimension in this study. We defined a rule to consider multiple builds on one OS and environment as failing if at least 50% of its builds failed or had unknown build results, which could impact our results. Future work can analyse in more details all these builds with multiple OS versions/architectures.
- Similar to recent work on Travis CI [14], we only chose those modules which were best fit for the study, so our findings may differ with a different dataset.

REFERENCES

[1] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[2] 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.

[3] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 2016, pp. 426–437.

[4] "Companiesusingci," https://www.quora.com/What-are-the-best-examples-of-\companies-using-continuous-deployment, accessed: 2017-09-20.

[5] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: a case study (at google)," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 724–734.

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

[7] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proceedings of the 33rd international conference on software engineering*. ACM, 2011, pp. 141–150.

[8] R. Hardt and E. V. Munson, "Ant build maintenance with formiga," in *Proceedings of the 1st International Workshop on Release Engineering*. IEEE Press, 2013, pp. 13–16.

[9] S. I. Feldman, "Make—a program for maintaining computer programs," *Software: Practice and experience*, vol. 9, no. 4, pp. 255–265, 1979.

[10] A. E. Hassan and K. Zhang, "Using decision trees to predict the certification result of a build," in *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 2006, pp. 189–198.

[11] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 235–245.

[12] "John oduinn," https://oduinn.com/2016/06/26/distributed-er8-now-available, accessed: 2016-12-07.

[13] "CPAN comprehensive perl archive network," http://www.cpan.org, accessed: 2015-12-22.

[14] 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*. IEEE Press, 2017, pp. 356–367.

[15] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, p. 122, 2006.

[16] "Travisci," https://travis-ci.org, accessed: 2017-09-20.

[17] "Stride," https://github.com/Strider-CD/strider, accessed: 2017-09-20.

[18] "Jenkin," https://jenkins.io, accessed: 2017-09-20.

[19] C. Watters and P. Johnson, "Version numbering in single development and test environment," Dec. 29 2011, uS Patent App. 13/339,906.

[20] "Buildbot," http://buildbot.net, accessed: 2017-09-20.

[21] blazemeter, "jenkins-vs-other-open-source-continuous-integration-servers." IEEE Press, 2017.

[22] "treeherde," https://treeherder.mozilla.org/#/jobs?repo=mozilla-inbound, accessed: 2017-09-20.

[23] "treeherder," https://treeherder.mozilla.org/#/jobs?repo=mozilla-inbound&revision=4f06cc3d4f39b4ff431792e2e14909a2b7655442, accessed: 2016-12-07.

[24] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 805–816.

[25] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö, "The highways and country roads to continuous deployment," *IEEE Software*, vol. 32, no. 2, pp. 64–72, 2015.

[26] E. Laukkanen, M. Paasivaara, and T. Arvonen, "Stakeholder perceptions of the adoption of continuous integration–a case study," in *Agile Conference (AGILE), 2015*. IEEE, 2015, pp. 11–20.

[27] G. V. Neville-Neil, "Kode vicious system changes and side effects," *Communications of the ACM*, vol. 52, no. 4, pp. 25–26, 2009.

[28] R. Adams, W. Tichy, and A. Weinert, "The cost of selective recompilation and environment processing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 3, no. 1, pp. 3–28, 1994.

[29] Y. Yu, H. Dayani-Fard, J. Mylopoulos, and P. Andritsos, "Reducing build time through precompilations for evolving large software," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on.* IEEE, 2005, pp. 59–68.

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

[31] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter, "Design recovery and maintenance of build systems," in *2007 IEEE International Conference on Software Maintenance.* IEEE, 2007, pp. 114–123.

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

[33] S. McIntosh, B. Adams, and A. E. Hassan, "The evolution of ant build systems," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010).* IEEE, 2010, pp. 42–51.

[34] G. Schermann, J. Cito, P. Leitner, U. Zdun, and H. Gall, "An empirical study on principles and practices of continuous delivery and deployment," PeerJ Preprints, Tech. Rep., 2016.

[35] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on.* IEEE, 2014, pp. 41–50.

[36] T. Rausch, W. Hummer, P. Leitner, and S. Schulte, "An empirical analysis of build failures in the continuous integration workflows of java-based open-source software," in *Proceedings of the 14th International Conference on Mining Software Repositories.* IEEE Press, 2017, pp. 345–355.

[37] N. Kerzazi and B. Adams, "Botched releases: Do we need to roll back? empirical study on a commercial web app," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 574–583.

[38] P. Denny, A. Luxton-Reilly, and E. Tempero, "All syntax errors are not equal," in *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education.* ACM, 2012, pp. 75–80.

[39] G. Dyke, "Which aspects of novice programmers' usage of an ide predict learning outcomes," in *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, 2011, pp. 505–510.

[40] A. Miller, "A hundred days of continuous integration," in *Agile, 2008. AGILE'08. Conference*. IEEE, 2008, pp. 289–293.

[41] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. Di Penta, and S. Panichella, "A tale of ci build failures: an open source and a financial organization perspective."

[42] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining co-change information to understand when build changes are necessary," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 241–250.

[43] M. F. Zibran, F. Z. Eishita, and C. K. Roy, "Useful, but usable? factors affecting the usability of apis," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*. IEEE, 2011, pp. 151–155.

[44] J. Dietrich, K. Jezek, and P. Brada, "Broken promises: An empirical study into evolution problems in java programs caused by library upgrades," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 64–73.

[45] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 70–79.

[46] W. Wu, F. Khomh, B. Adams, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of api changes and usages based on apache and eclipse ecosystems," *Empirical Software Engineering*, pp. 1–47, 2015.

[47] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, 2017.

[48] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, pp. 1–34, 2017.

[49] D. Dig and R. Johnson, "How do apis evolve? a story of refactoring," *Journal of software maintenance and evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006.

[50] 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 *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 312–322.

[51] "metacpan-api," https://github.com/metacpan/metacpan-api, accessed: 2016-12-07.

[52] R. O. Rogers, "Scaling continuous integration," in *International Conference on Extreme Programming and Agile Processes in Software Engineering*. Springer, 2004, pp. 68–76.

[53] Y. Jiang, B. Adams, F. Khomh, and D. M. German, "Tracing back the history of commits in low-tech reviewing environments: a case study of the linux kernel," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014, p. 51.

[54] M. B. Miles and A. M. Huberman, *Qualitative data analysis: An expanded sourcebook*. sage, 1994.

[55] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 international conference on software engineering*. IEEE Press, 2013, pp. 712–721.

[56] H. Hemmati, S. Nadi, O. Baysal, O. Kononenko, W. Wang, R. Holmes, and M. W. Godfrey, "The msr cookbook: Mining a decade of research," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE, 2013, pp. 343–352.

[57] M. Shridhar, B. Adams, and F. Khomh, "A qualitative analysis of software build system changes and build ownership styles," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014, p. 29.

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

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

[60] cpan@perl.org, "PerlSource versions and release date," accessed: 2016-11-01. [Online]. Available: http://www.cpan.org/src/

[61] M. M. Lehman, "Laws of software evolution revisited," in *European Workshop on Software Process Technology*. Springer, 1996, pp. 108–124.

[62] R. R. Bouckaert, E. Frank, M. Hall, R. Kirkby, P. Reutemann, A. Seewald, and D. Scuse, "Weka manual for version 3-7-3," *The university of WAIKATO*, 2010.

[63] M. L. Calle, V. Urrea, A.-L. Boulesteix, and N. Malats, "Auc-rf: a new strategy for genomic profiling with random forest," *Human heredity*, vol. 72, no. 2, pp. 121–132, 2011.