Noname manuscript No.
(will be inserted by the editor)

A Study of Build Inflation in 30 Million CPAN Builds on 13 Perl Versions and 10 Operating Systems

Mahdis Zolfagharinia · Bram Adams · Yann-Gaël Guéhéneuc

Received: date / Accepted: date

Abstract Continuous Integration (CI) is a cornerstone of modern quality assurance, providing on-demand builds (compilation and tests) of code changes or software releases. Yet, the many existing CI systems do not help developers in interpreting build results, in particular when facing build inflation. Build inflation arises when each code change has to be built on dozens of combinations (configurations) of runtime environments (REs), operating systems (OSes), and hardware architectures (HAs). A code change C1 sent to the CI system may introduce programming faults that result in all these builds to fail, while a change C2 introducing a new library dependency might only lead one particular build configuration to fail. Consequently, the *one* build failure due to C2 will be "hidden" among the *dozens* of build failures due to C1 when the CI system reports the results of the builds. We have named this phenomenon *build inflation*, because it may bias the interpretation of build results by developers by "hiding" certain types of faults.

In this paper, we study build inflation through a large-scale study of the relationship between REs and OSes and build failures on 30 million builds of the CPAN repository on the CPAN Testers package-level CI system. We show that the builds of Perl packages may fail differently on different REs and OSes and any combination thereof. Thus, we show that the results provided by CPAN Testers require filtering and selection to identify real trends of build failures among the many failures. Manual analysis of 791 build failures shows that dependency faults (missing modules) and programming faults (undefined values) are the main reasons for failures, with dependency faults being easier to fix. We conclude with recommendations for practitioners and researchers in interpreting build results as well as for tool builders who should improve the scheduling of builds and the reporting of build failures.

Keywords Continuous integration, build inflation and failure, Perl, CPAN

MCIS and Ptidej labs, Polytechnique Montréal
Québec, Canada
E-mail: mahdis.zolfagharinia,bram.adams,yann-gael.gueheneuc@polymtl.ca

1 Introduction

Continuous integration (CI) systems are important for quality assurance [1] by building and testing each commit entering the review environment or the version control system of an individual project, as well as final or intermediate releases as they are submitted to an ecosystem (e.g., Linux distribution). They combine build and test scripts to run compilers and other tools and to test the compiled code [2, 3] and notify developers of build failures [4, 5].

CI systems perform many different build-related tasks. For example, open-source projects like OpenStack [6] or Mozilla [7] use CI systems for experimental ("try") builds before a patch (a pull request) is sent out for review, integration builds after acceptance of a patch, builds to prepare and perform a new release, builds for static analyses, etc. CI systems must schedule builds on configurations of runtime environments (REs), operating systems (OSes), and hardware architectures (HAs) identical or as close as possible to their production counterparts [8, 9]. Nowadays, configuration and instantiation of the REs, OSes and HAs for a given build can easily be automated using Infrastructure-as-Code (IaC) languages like Chef or Puppet, and deployment technologies like virtual machines or containers.

The many possible, legitimate configurations of REs, OSes, and HAs create a phenomemon of *build inflation* in CI systems: each single commit yields a large number of builds, one build per configuration. This build inflation has several negative consequences on organizations, developers, and researchers. For organizations, performing more builds strains the CI systems (for example at Google [10]). This strain can be absorbed by a cloud infrastructure, but at a cost: in 2013, O'Duinn, Mozilla's former head of release engineering, estimated that the build cost per commit was $26.40 [11], on average, for a total of about $201,000 in December 2013 (7,601 commits × $26.40).

Performing more builds makes the interpretation of their results more complex. For example, commit `e17e25c` of the Ruby on Rails project led to 42 builds, one of which failed because an API was not supported by Ruby 1.8.7. As a result, the commit was marked as a "failing build" and displayed as such by Travis CI. Developers then had to decide whether one build failure out of 42 warranted the withdrawal of the commit or not: were all the 41 successful builds equally useful as was the unique failing build? Would it be useful to add a 43^{rd} build and, if so, with what configuration of REs, OSes, and HAs?

For researchers, performing more builds provides a wealth of data unavailable in the past about a vast array of diverse software projects. However, researchers are concerned whether this data is actually representative of different phenomena or instead pertains to the same phenomenon repeated over and over again. They are also concerned with the costs, for organizations and developers, of configuring, instantiating, using, and maintaining large CI systems, with many configurations of REs, OSes, and HAs, and want to reduce these costs by understanding build inflation.

Consequently, this paper studies 30 million builds of one CI system to observe and report whether build inflation exists, its impact on builds, and

provide recommendations to organizations, developers, and researchers. We choose to study CPAN Testers, the CI system of the Comprehensive Perl Archive Network (CPAN) [12], which is the official repository of Perl packages, because CPAN Testers builds CPAN libraries across a wide range of environments, such as different Perl REs, OSes, and HAs. This provides the data required to study build inflation. We analyze the build data generated between 2011 to 2016, covering more than 12,000 CPAN packages, 13 Perl RE versions, and 10 OSes—the largest quantitative observational study of builds—to answer the following research questions:

- RQ1: How do build failures evolve across time?
- RQ2: How do build failures spread across OSes and Perl versions?
- RQ3: How do build failures relate to Perl versions?
- RQ4: How do build failures relate to OSes?
- RQ5: What are the different types of build faults?
- RQ6: How do build fault types relate to OSes?
- RQ7: How do build fault types relate to Perl versions?

By answering these questions, we show an inflation in the numbers of builds and build failures, which hides the reality of builds in noise. For example, comparing millions of builds on Linux with thousands of builds on Cygwin may lead to wrong conclusions about the quality of Perl versions on Cygwin in comparison to Linux. We also show that unnecessary builds, e.g., certain OSes/Perl RE versions, for which we already have many builds, could be avoided altogether. Thus, we provide empirical evidence of the costs of build inflation. We conclude by providing recommendations to organizations/developers and researchers to deal with this inflation.

This paper extends our previous work [13] with an analysis of the types of build faults and of the relations between build fault types and REs and OSes, respectively. Consequently, we added RQ5, RQ6, and RQ7 to describe each of these types and relations. Section 2 also adds a detailed comparison of the types of build faults with those reported in the literature.

This paper is organized as follows: Section 2 presents background information on CPAN Testers and major related work. Section 3 describes our observational study design, while Section 4 presents our observations, followed by discussion and recommendations in Section 5. Section 6 describes threats to the validity of our study. Finally, Section 7 concludes with future work.

2 Background and Related Work

2.1 CPAN and CPAN Testers

Similar to Maven and npm for Java and Node.js, the Comprehensive Perl Archive Network[1] (CPAN) [12] is a repository of *modules* that Perl developers can require and install. These modules provide various applications and

[1] `http://www.cpan.org`

libraries, e.g., `DBI` to connect to databases or `JSON` to encode/decode in the JSON format. CPAN currently contains more than 255,000 modules bundled into 39,000 *packages*[2], published by independent contributors. Each package combines the set of modules with their documentation, tests, build and installation scripts. A package can have one or more *versions*.

Developers can add additional packages to their local Perl RE through the `cpan` command line tool. When a developer asks it to install a package version, it downloads that version from the nearest CPAN mirror, unpacks, processes, and–or transforms code and data, compiles any native C code, and runs its unit tests. Since there is no official Perl mechanism for developers to share their internal build results with users, users and other developers can consult CPAN Testers [14] to understand whether a package version is likely to build successfully in their own REs, *before installation*. As such, CPAN Testers allows developers (1) to share their local build results along with the exact REs in which they were obtained and (2) to visualize build results per package version, RE (e.g., Perl version 5.8 vs. 5.19), OS (e.g., Perl version 5.8 vs. 5.19) and even HA (e.g., Perl version 5.8 vs. 5.19).

The CPAN Testers infrastructure differs from other CI systems, like Travis CI and Jenkins, by its granularity and architecture. CPAN Testers provides build results per package version rather than per commit, since it is related to the Perl ecosystem rather than to one individual project's development process. Furthermore, CPAN Testers is a distributed CI system, because its build machines depend on contributions of volunteering developers. Consequently, CPAN Testers cannot guarantee that every package version will be built on every configuration of REs, OSes, and HAs. A similar phenomenon exists for commit-level CI systems of large organizations, who are investing in just-in-time scheduling of CI builds [10] or in grouping of commits [15] because their (centralized) CI infrastructure is unable to cope with the influx of code changes to build. On the upside, CPAN Testers provides build results from configurations of REs, OSes, and HAs that are *used in practice* instead of *artificial* configurations maintained in-house.

Figure 1 shows the overview page of the build results of version 0.004002 of `List-Objects-Types` for Perl RE versions 5.8.8 to 5.19.3 (left column) and OSes CygWin to Solaris (top row). Each cell aggregates all build results for a given configuration of RE and OS across all considered HAs. Red cells indicate that all builds (across all HAs) for a given configuration of RE and OS failed, green cells that all builds were successful, and red/green cells that some builds (HAs) failed. White cells indicate missing build results and orange cells unknown results. CPAN Testers also provides the number of successful, failing, and unknown build results. Unknown build results comprise 4% of the build results and typically correspond to builds or tests that were interrupted before any output could be generated. In RQ3 and RQ4, we consider unknown results as failures, because they are not successful build results. In RQ5, RQ6

[2] In this paper, we use the term "package" in its usual sense, while Perl developers would talk about "distribution".

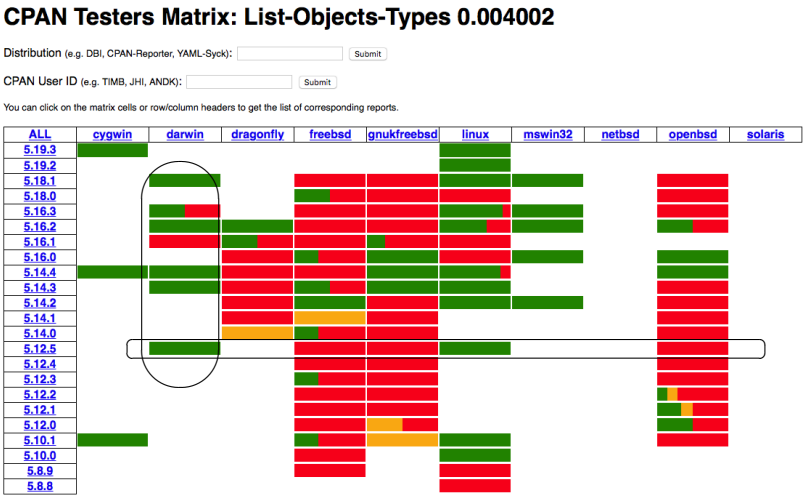

Fig. 1 Example CPAN Testers build report. A vertical ellipse represents a "RE build vector" (RQ3); a horizontal one represents an "OS build vector" (RQ4).

and RQ7, we ignore unknown results altogether, because their build logs do not provide any information about the causes of build failures.

2.2 Related Work

2.2.1 (CI) Builds

While build scripts exist in one form or another since the early days of software development, typically as regular batch scripts, dedicated build script languages like Make [16] originated at the end of the '70s. While modern Make incarnations still exist, current developers can choose amongst dozens of different build technologies [17], such as Ant, Maven, Gradle, Bazel, Rake, SCons and CMake. CI systems date back to 1991, when Booch promoted continuous integration in his development method [18]. The advent of extreme programming and (more recently) continuous delivery [8] has sparked a wide range of CI technologies both in the strict (commit-level) sense, such as Bamboo, Hudson, Jenkins, TeamCity, and Travis CI, and in the wider (release-level) sense, such as CPAN Testers.

Both build scripts and CI systems have been amply studied in the literature. Tu et al. [19] proposed the concept of build-time views, which are architectural views [20] that represent the run-time flow of the (GNU Make) build process of a software system. Adams et al. [2, 21] analyzed the evolution of the Makefiles that form the Linux kernel build system and reported that these Makefiles grow in number and size over time and require frequent changes, a trend that was confirmed by McIntosh et al. [22] for Ant files. Later, McIntosh et al. [23, 24] empirically showed that, in general, build scripts and

source code co-evolve and hence require a non-trivial maintenance effort from developers. Suvorov et al. [25] study how 2 open source projects, Linux and KDE, migrated their build scripts between different technologies. More recent work on build scripts focused on errors in build scripts, since these are tricky to detect and fix. Macho et al. [26] and Hassan et al. [27] proposed techniques to automatically repair build dependencies and build script instructions.

While this previous work shows that build scripts are important software artifacts that require effort to evolve, debug and fix, more recently the research community turned towards the use of these build scripts to power CI activities. Vasilescu et al. [28] conducted an empirical study about CI usage among 246 projects in GitHub and showed that CI significantly improves the productivity of GitHub teams. Hilton et al. [29] assessed 34,544 open-source projects in GitHub, 40% of which use CI and reported that CI can help projects to release regularly. Three other works performed case studies on the use of CI. Miller [30] and Leppanen et al. [31] found a positive impact of CI in terms of accelerated delivery of value to customers (up to 40% speedup [30]), while Laukkanen et al. [32] identified several challenges of CI adoption, such as the quality of test cases, learning difficulties, and need for architectural changes.

In practice, continuous integration at commit level means that builds are triggered very often and, thus, CI must be efficient. In practice, the high volume of commits as well as the inflation of builds due to the many configurations of REs, OSes, and HAs that need to be built and supported, make commit-level CI intractable. This is why organizations like Google [33] and OpenStack [15] started to adopt build clustering approaches that basically buffer all new commits that arrived in a particular time interval, then release the cluster as a whole to CI. While build successes significantly reduce the build time by the number of commits in the cluster, any build failure requires a tedious root cause analysis to determine the culprit commit(s). Reducing the time for such root cause analysis is the focus of ongoing research [15, 33].

2.2.2 Build Failures

The domain of analysis and prediction of build failures and their causes (build faults) grew with the availability of large build data sets. Table 1 shows an overview of the different causes of build failures unearthed by existing work. Hassan et al. [34] modeled build results in terms of developers' work habits, team size, developers' experience, change complexity, integration interactions, and previous build results; with the latter being the best indicator of future build results.

Miller [30] studied 66 build failures in Microsoft projects and categorized their causes into faults related to compilation (26%), unit testing (28%), static analysis (40%), and server issues (6%). Dyke [35] assessed the frequency of compilation errors by tracking Eclipse IDE usage with novice developers. Denny et al. [36] investigated compilation errors in short snippets of Java code and showed that 48% of the builds failed due to compilation errors. Our

Table 1 Comparison of build fault types and other factors found by related work to be significant indicators of build failures. The numbers in a column for a particular paper indicate the ranking of build fault types in that paper. An "x" indicates a factor that was not ranked, but which the paper spent considerable attention on.

Failure type	[34]	[30]	[35]	[36]	[3]	[37]	[38]	[4]	[39]	[40]	[41]	this paper
syntactic compilation errors		2	x	x	x		1		9	x	3	x
semantic compilation errors					x							
risky part of code	2											
internal code deps.					x							
external code deps.						x	2		8	x	5	x
static analysis failures		1					3		4		6	
unit test failures		3		x			1	1	1	x	2	x
integration test failure							4		5		5	
crosscutting tests									3			
missing files in commit						x			2	x		
incorrect code committed						x						
type of code change						3						
scope of code change						6	x					
file type changed							x					
unstable code							x					
documentation issue							5				8	
license issue							6					
dev. experience							x					
bad coder	2											
role of developer						1					x	
size of dev. team						2						
previous build failed	1						x					
type of build						4	x					
build script error							2				4	
external build tools									7			
git interaction error							1		2			
build environment issues		4					2			x	1	
remote deployment site									6		7	
timing vs. release cycle						5						
explicitly skipped tests										x	x	
passively skipped failures											x	
RE											x	x
OS											x	x

paper instead studies 791 build failures in the context of release-level CI and proposes a finer-grained fault categorization.

Seo et al. [3] performed the first large-scale analysis of build failures by analyzing 26.6 million builds on C and Java projects at Google. Focusing on build failures caused by compilation problems, they concluded that (1) undeclared or missing variables, methods, and classes are the main sources of compilation errors, often caused by missing packages, (2) the time to fix these errors varies widely, and (3) developers need tool support to avoid and interpret these errors. While focusing on build failures in general (not just compilation errors) and Perl, our paper confirms that programming errors and missing external dependencies are common sources of build failures. In

addition, we also consider the impact of different REs and OSes on the types of build faults.

Kerzazi et al. [37] studied 3,214 builds made during a 6-month period in a large web company and reported that 17.9% of the build failures have a potential cost of about 2,035 man-hours. They identified different fault types with the most common failures caused by missing files in pull requests, accidental check-in of experimental changes, missing specifications of transitive dependencies, branch merges, larger teams, and the time of the builds relative to the release cycle. We study build faults in open-source Perl packages rather than a closed-source web application. Apart from dependency issues, none of the factors studied by Kerzazi et al. are applicable to release-level CI.

Ståhl and Bosch [42] surveyed CI practices and build failures, and reported that test failures during builds are sometimes accepted by developers because they know that these particular failures will be fixed later [43]. These implicitly-accepted failures make interpretation of build results non-intuitive, because CI systems do not distinguish such failures from others. Similar to this paper, we consider a phenomenon, build inflation, that reduces the signal-to-noise ratio of build results, making it hard to draw correct conclusions.

Rausch et al. [38] analyzed Travis CI build failures in 14 open-source Java projects. Similar to Hassan et al. [34], they found that prior build results, change complexity, and developers' experience are the best indicators for future build results. They found that more than 80% of the failures are due to test failures, code quality issues, compilation errors, git access errors, and build script errors. Other failures are due to build crashes, dependency errors, integration test failures, documentation errors, API incompatibility, and Android SDK issues. We confirm that more than 80% of the failures are related to compilation errors and missing packages (i.e., dependency errors), but also to OSes and REs, not considered by Rausch et al. or Hassan et al.

Beller et al. [4] conducted an 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 have 10 times more tests and hence have a higher build failure ratio than Java projects. In parallel with our earlier work [13], they observed that builds on different REs are being run for each commit, which leads to inconsistent build results. In particular, for more than 60% of the projects, at least one build had different results across different REs. Our paper investigates this observation in depth, considering REs and OSes and performing both qualitative and quantitative analyses of build results.

Vassallo et al. [39] studied build failures in 418 Java projects at a company and 349 GitHub projects (that use Travis CI). They grouped build failures across the different build phases of the Maven build system, yielding 20 categories. They mapped each category to keywords in order to automatically classify the 34,182 build failures. They showed that the open-source projects had a much larger percentage of unit testing failures (28% vs. 5.2%) than the company's projects, while the company's build failures had more failures related to git interaction errors and missing files (21.1% vs. 0.0%), integra-

tion testing (13.3% vs. 5.0%), other kinds of testing (18.3% vs. 8.3%), remote deployment sites (10.0% vs. 0.5%), and external build tools (8.8% vs. 1.4%). Instead of grouping build faults by build phase, we classify them based on the semantics of the build fault.

Zhao et al. [40] performed quantitative and qualitative analyses of thousands of GitHub projects in 7 programming languages that migrated to Travis CI. They found that more code is being built and tested per build and more pull requests are successfully closed, even though the requests take longer to be closed. They also observed an increase in the number of unit tests per build. Through open coding, they find an upward trend of build failures due to compilation errors, execution errors, failed tests, and skipped tests, while missing dependencies and time-outs see a downward trend with prolonged CI usage. They did not consider failures caused by REs and OSes.

Finally, the closest work to this proposal is the recent analysis of 3.7 million GitHub build jobs by Gallaba et al. [41], which was published while the first revision of this paper was under review. Building on our MSR 2017 study [13], they found that 12% of passing CI builds actually contain failing build jobs or build jobs explicitly marked to be skipped (i.e., ignored by the CI system). Furthermore, 2 out of 3 build failures occur for more than one CI run in a row. This indicates that developers implicitly ignored them, for example because they know someone else will be working on them [42]. These findings again indicate the presence of noise in CI results. Furthermore, the authors found additional empirical proof of build inflation and of build failures, such as the fact that 44% of the studied build failures were RE-dependent. Gallaba et al.'s empirical evidence on build failure noise and heterogeneity complements the results of our current and earlier work on the effects of build inflation.

To conclude, the 34 causes and indicators of build failures in Table 1 cover a wide range of dimensions. The most commonly found causes of build failures are syntactic errors (9 studies), unit test failures (8 studies) and external code dependencies (6 studies). While the other discussed papers identified these top fault types for commit-based CI systems, we confirm their presence in release-based CI systems. Together with Gallaba et al. [41], we are the only work considering the impact of REs and OSes on build failures.

3 Observational Study Design

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

3.1 Study Object

The object of our study is the relationship between build results and REs and OSes when facing build inflation. Such inflation refers to the phenomenon of excessive numbers of builds, caused by the many configurations of REs and

OSes, that introduces bias in build results and that leads to incorrect interpretations of the results by developers. For example, one failed Windows build for a Perl release built on 99 Linux machines and one Windows machine has a different impact than 50 failed Windows builds on 50 Windows and 50 Linux machines. Similarly, build failures that occur across all REs are different to failures happening only on one specific RE. Based on our findings, organizations/developers and researchers would need to consider build inflation caused by multiple REs and OSes, e.g., by considering the specific numbers of build failures per RE and OS instead of their total number, as currently provided by most CI systems. We provide more recommendations in Section 5.

3.2 Study Subject

We choose CPAN Testers to study the relationship of Perl REs and OSes with build failures because it provides the build results of *all* Perl packages and their releases on dozens of Perl REs, OSes and HAs. An alternative subject could be Mozilla TreeHerder [7], which provides centralized, commit-level CI results on several configurations of REs, OSes and HAs, but for a smaller number of projects. We could not use the TravisTorrent data set [4] because the projects using Travis CI system almost exclusively build on the default Ubuntu OS (although they do consider multiple versions of the Java RE).

The reason why CPAN Testers provides build data across a diverse range of REs and OSes is because, contrary to most CI systems, it consists of a heterogeneous grid of machines managed by volunteers. Basically, by installing a daemon on one's machine, it becomes part of the CPAN Testers build grid, sending build and test reports to a central machine. Since this makes the process of joining and leaving lightweight, the composition of the build grid frequently changes. On the upside, this provides a very rich environment of REs, OSes and HAs for our study.

We use CPAN Testers' REST API [44] and crawl the CPAN web site [12] to collect the build logs and meta-data of all package versions. Build logs contain build results as well as the executed commands and any generated error messages. The `META.yml` meta-data files contain the package names, versions, dependencies, authors, and other information (e.g., supported Perl REs and OSes). We collect all build logs and meta-data between January 2000 and August 2016 as data set for our observational study, which includes 68.9 million builds for 39,000 packages, 103 REs, and 27 OSes. As such, this data set spans a longer time period than TreeHerder or Travis CI.

3.3 Quantitative Study Sample

We obtain our study sample as follows. Analysis of the complete data set of 68.9 million builds shows that most builds were performed between 2011 and 2016, with 13 top REs (Perl 5.8 to 5.21, excluding 5.9) and 10 top OSes. Each

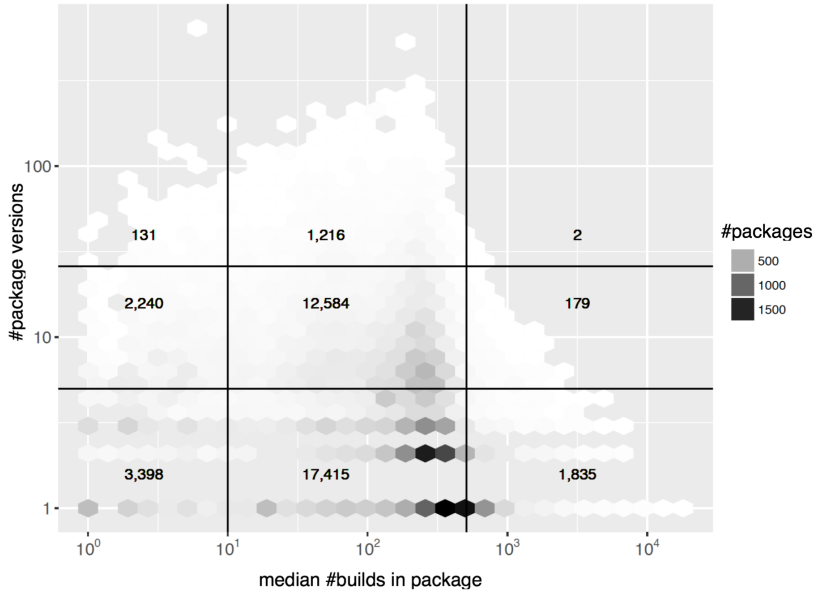

Fig. 2 Hexbin plot of the build results (darker cells). Each cell represents the number of CPAN packages with a given median number of builds (x-axis) and of versions (y-axis). The black lines correspond to the thresholds used to filter the build results, dividing the data set into 9 quadrants. Each quadrant shows the number of included packages. The central quadrant contains the data set used in our study.

of these 13 REs had more than 800,000 builds, and each of the 10 OSes had more than one million builds. In order to simplify our analyses and discussions of results, we defer the analysis of HAs to future work. Consequently, we focus on the 62.8 million builds performed by these 13 REs and 10 OSes between January 2011 to June 2016, reducing the number of builds by only 8.85%, and REs by 87.4% and OSes by 63%.

Figure 2 illustrates the distributions of the median number of builds and of versions for all packages in our data set. It also shows the number of packages within each quadrant. We notice a large variance in both the number of versions and the number of package versions. In addition to these numbers, we found for example that 13,522 packages have more than 1,000 builds across all their versions, while 967 have fewer than 3 builds. Conversely, some packages have build results for only few of their versions. Packages with too few builds or too few versions would skew our data set because they do not provide enough build results to analyze. Therefore, we exclude such packages. Similarly, we also exclude packages with too many builds or versions.

In order to filter our data set, we determine lower and upper thresholds for the number of package versions and number of builds per package by considering the density of points in Figure 2. We choose the lower threshold for the median number of build results of a package as 10 and of a package version as 5. We use the inter-quartile range of the data to compute the upper thresholds

using the approach suggested by Wohlin et al. [45]: $ut = (uq - lq) * 1.5 + uq$, where lq and uq are the 25^{th} and 75^{th} percentiles, which yields 509 as upper threshold for packages and 26 for package versions. After removing lower and upper outliers, we obtain a final data set of 30 million builds for the 12,584 CPAN packages, corresponding to the central quadrant in Figure 2.

In order to study the relation between build failures and REs (RQ3 and RQ7) or OSes (RQ4 and RQ6), we abstract up the build data in the build result matrix (Figure 1) of each package version by considering vertical rows (RQ3/7) or horizontal rows (RQ4/6) of build results. Vertical rows are called "RE build vectors", while horizontal rows are "OS build vectors".

The content of these vectors, i.e., the values used for each cell in the matrix, depends on the specific RQ. In RQ3 and RQ4, the vectors consider only the most common build result. For example, if the majority of builds for a given configuration of RE and OS were successful, we put "succes" in the corresponding vector element. In RQ6 and RQ7, we consider only the most recent build failure for each configuration of RE and OS, as explained in the next subsection. Note that RQ1 and RQ2, which do not use the concept of vectors, consider respectively all possible build results for a given configuration of RE and OS (RQ1: 100% success/fail, mixed success, unknown outcome or missing outcome), or all results minus the unknown outcomes (RQ2).

3.4 Qualitative Study Sample

For RQ5, RQ6 and RQ7, we perform a qualitative study of build failures to categorize the different fault types and to understand the relationships between these fault types and REs/OSes. We manually analyze the build logs to identify the fault types and to compare their relative frequencies. Across all configurations of RE and OS of all analyzed package versions, we observe a median number of 64 build failures, with a maximum of 1,362 failures. Since the build failures of a package version for a given configuration of RE and OS have a high probability of being caused by similar faults, we consider only the most recent failure for each configuration of RE and OS in these research questions.

In RQ6, we are interested in possible dependencies between build faults and OSes, we collect, for each configuration of RE and OS containing a build failure, its entire OS build vector to analyze whether the same failure also occurred in other OSes for the same RE and package version. We only consider OS build vectors containing build results for at least 10 OSes. Similarly, for RQ7, we extract the RE build vector of each failure.

We split the OS build vectors into "minority vectors" (one to three failing OSes) and "majority vectors" (six to ten failing OSes), while ignoring vectors with four or five failing OSes to understand the differences between inconsistent (minority vectors) and consistent build failures (majority vectors). The resulting OS vectors cover 1,421 build failures (out of 76,748 in the full data

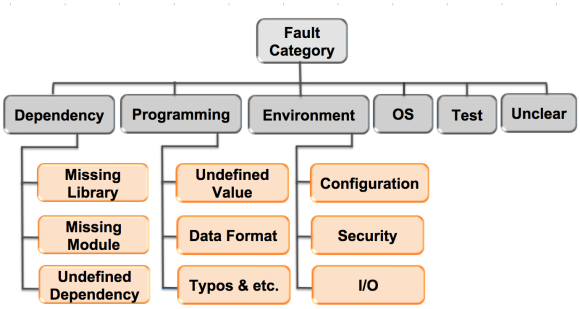

Fig. 3 Hierarchy of fault types across all OSes and REs.

set), spread across 804 OS build vectors[3]. We group the vectors into 752 minority vectors (963 failures) and 52 majority vectors (458 failures).

For manual analysis, we select a random sample of 791 failures among the 1,421 failures, with a confidence level of 95% and a confidence interval of 5%. This sample contains 52 majority vectors with 458 failures and 254 minority vectors with 333 failures. In a first iteration, the first author extracts the error messages in the build results for each failure, explores on-line reports and feedback, and identifies the root faults of the failures. She collects the error messages and their root faults into cards in Google Keep. She groups these cards when they share similar root faults. In the second and third iterations, the first and second author revisit each category and, using open coding, discuss any differences among root faults. Major reasons for differences/disagreements are (1) unclear error messages and (2) too broad/narrow categories. This "negotiated agreement" [46, 47] led to a consensus of 6 categories and 9 subcategories of failure types, shown in Figure 3 and studied in RQ5, RQ6, and RQ7.

4 Observational Study Results

We now present the motivations, approaches, and results of RQ1 to RQ7.

RQ1: How do build failures evolve across time?

Motivation. With this research question, we want to understand how often builds fail in CPAN Testers and whether the ratio of failing builds is constant or changes over time. Beller et al. [4] reported failure ratios of 2.9% and 12.7% for Java and Ruby builds, respectively, in Travis CI; Seo et al. [3] of 37.4% and 29.7% for C++ and Java builds at Google. They consider only commit-level build results and did not study the evolution of these ratios over time.

[3] Most of the vectors did not contain any build failure, which is expected.

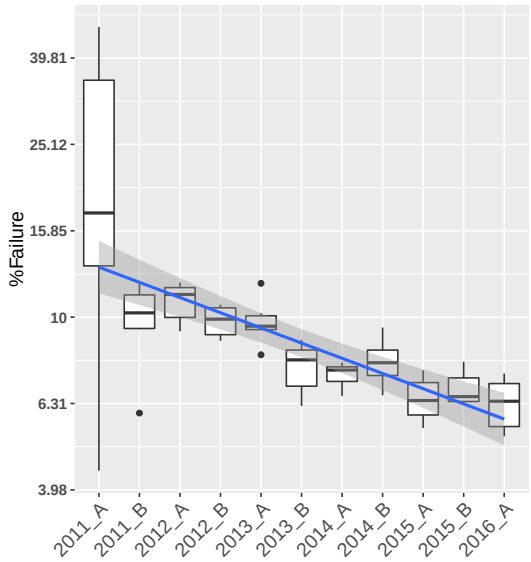

Fig. 4 Distribution of failure ratios in 6-month periods, with linear regression showing the trend of the ratios over the studied 6 years.

Table 2 Number of builds, package versions, and average number of builds per package version in each 6-month period 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
#builds	626	946K	1,860K	2,404K	3,021K	3,482K	3,625K	4,082K	4,827K	3,394K	2,891K
#package versions	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

Approach. We consider as build failures all failing and unknown build results. For each CPAN package in the data set of 30 million builds, we compute its ratio of build failures as $\#buildfailures/\#builds$ (ignoring the different versions of packages). We investigate the evolution of failures per period of 6 months because every 6 months a new Perl RE version was released between 2010 to 2014. We do not distinguish between REs and OSes in this RQ.

Findings. *The median build failure ratio decreases across time from 17.7% in the first 6 months of 2011 to 6.3% in the first 6 months of 2016.* Figure 4 shows the distribution of the failure ratios of all package builds in the studied 6 years. Between 2011 and 2013, the median failure ratio in the first half of the year is higher than that of the second half. This trend reverses from 2014. The regression line shows that build failure ratios decrease between 2011 and 2016, especially when considering the logarithmic scale used in the figure. We

explore 2 hypotheses to explain the decreasing failure ratio trend: (1) fewer builds performed over time or (2) fewer package versions released over time, reducing the probability of build failures.

The number of builds increased by a factor of 3 to 5 from the second half of 2011 on. Table 2 shows the number of builds per period of 6 months: even though more builds are performed over time, they also are more successful over time, i.e., there is an inverse correlation between the number of builds and of build failures. For comparison, Atlee reported a 6-fold increase in the number of builds of Mozilla Firefox between November 2009 and September 2013 [48]. While we cannot explain the decreasing number of builds in the second half of 2015 and the first half of 2016, this decrease is responsible for the plateau (instead of decrease) of median values for the rightmost box-plots in Figure 4.

The average number of builds per package version shows a 10-fold increase over time from 44.7 to 412.9, with some fluctuations from the second half of 2014 on (i.e., 2014-B). The average number of builds per version in Table 2 increases from 44.7 in the first 6 months of 2011 to 501.7 in the first half of 2015 with a slight dip at the end of 2014, after which it drops but still remains higher than in 2014. This observation can be explained as follows: the number of builds decreases from the second half of 2015, yet the number of versions did not decrease at the same rate.

While the increasing number of package versions is typical of today's rapid release strategies [49], the decreasing build failure ratio seems to be impacted much more by the 10-fold increase in the number of builds per package version. The next research question helps understand the impact of REs and OSes on this increase.

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

RQ2: How do build failures spread across OSes and Perl versions?

Motivation. We explain the decrease of the build failure ratio in RQ1 in terms of build inflation: each new package version is built multiple times, with most builds succeeding. Essentially, the same features are built and tested on every configuration of REs and OSes, such that feature-related faults result in failures in *all* REs and OSes, while RE- or OS-specific failures occur only for the (very) few problematic REs or OSes. For example, a Windows-specific fault would result in one build failure among dozens of successful builds on non-Windows OSes, a low build failure ratio giving a false impression of success. Beller et al. [4] suggested "to do continuous integration in several environments when their execution leads to different results, capturing errors that would not have been caught with one single environment".

Approach. For each build, we compute build failure ratios per RE and OS. As explained in Section 3.3, this RQ ignores builds with unknown results.

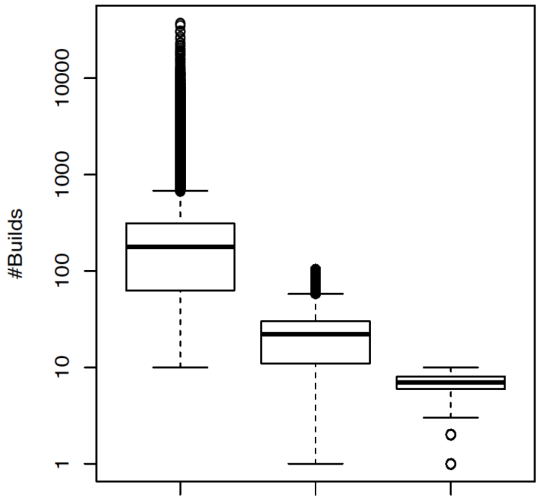

Fig. 5 Distribution of the number of (1) builds per package and of (2) REs and (3) OSes on which the builds took place.

Findings. *The package versions have a median of 179 builds on a median of 22 REs and 7 OSes.* Figure 5 shows the distribution of the number of builds, REs, and OSes across all packages. The distribution of OSes is more or less stable around 7 (low variance). However, the distribution of REs and especially of the number of builds per package have much higher medians and larger variance. There is a correlation between the product of the number of REs and OSes, and the number of builds per package.

We observe through a manual analysis of the data set that, when developers release a new package version, it is built on most of the REs and OSes to check whether it is backward compatible with their APIs [50]. Similarly, when a new RE or OS becomes available, most of the existing, non-deprecated package versions are re-built, which explains the increasing number of builds observed in RQ1, *but not* the decrease of the build failure ratio.

Not every RE yields equally representative build results. Figure 6 shows the evolution of build failures from Perl version 5.8 (released in 2002) to 5.21 (2015). REs are shown on the x-axis, ordered by release date [51], while the y-axis shows the build failure ratios (blue; right axis) and the percentages of builds of that package version on a given RE (black; left axis). The jagged trend of the percentages of builds suggests that odd releases are built substantially less than even ones.

Indeed, Perl version numbers have a fixed semantics [51]: even numbers, like 5.8, are official production releases (with maintenance releases, such as 5.12.1, for bug fixes) and odd numbers, like 5.11, are development releases. CPAN Testers prioritizes stable REs over development REs, which have consequently fewer, less representative, and less reliable build results. Only RE versions 5.19 and 5.21 had the same or less failures as their stable predecessors.

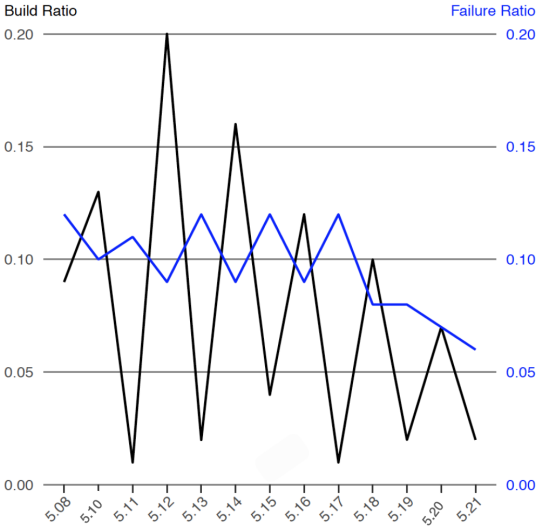

Fig. 6 Distribution of the ratios of all builds performed on an RE (black y-axis) and proportions of failing builds (blue y-axis).

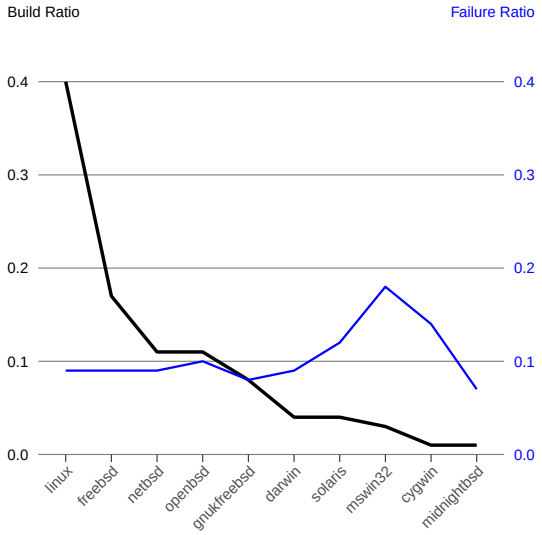

Fig. 7 Distribution of the ratios of all builds performed on an OS (black y-axis) and proportions of failing builds (blue y-axis).

Windows (18%), Cygwin (14%) and Solaris (12%) have substantially more build failures than other OSes. Figure 7 shows a clear difference between BSDs/Linux on the one hand and Windows/Cygwin/Solaris on the other in terms of the percentage of builds and build failures. The former cluster of OSes has a substantially larger number of builds than the latter while the

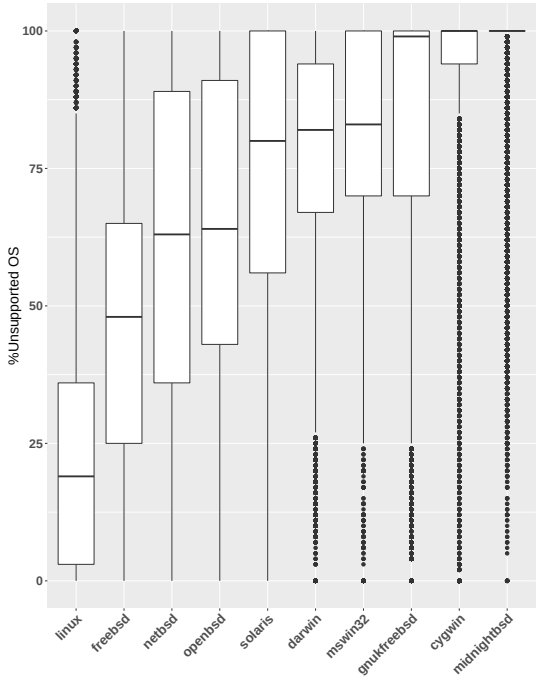

Fig. 8 For a given OS, distribution across package versions of the percentages of REs for which **no** builds were performed.

percentages of failures are lower. Similar to Perl versions, the build results on some OSes are less representative than others, because most Perl developers use Linux for development. Cygwin, Solaris, and Windows are less common OSes among Perl developers, yielding less builds and more failing builds.

We make similar observations by counting the number of times package versions are *not* built on configurations of REs and OSes, because no CPAN Testers volunteer has contributed such a machine configuration. Missing configurations correspond to empty cells in Figure 1 and their distributions for each OS are shown in Figure 8. The most incomplete OSes coincide with the OSes having most build failures in Figure 7. Therefore, the popularity of REs and OSes (and combinations thereof) among Perl developers impacts and could bias build results: build failures on less common configurations are drowned by successful builds on dozens of popular configurations.

RQ2: The environments (RE/OS) with the least builds have the highest proportion of failures, yet those numbers are drowned out by the larger (inflated) number of successes on more popular environments.

Table 3 Total percentage of the 4 patterns across OSes. "Pure" refers to occurrences of the patterns without fluctuation (e.g., $[1, 1, 1]$) while "Noisy" refers to occurrences with fluctuations (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%

RQ3: How do build failures relate to Perl versions?

Motivation. RQ2 provided evidence of build inflation due to popular configurations of REs and OSes. Yet, it does not explain why the build failure ratio decreases over time. We hypothesize that most of the build failures are specific to one RE and, hence, only count for one build failure compared to a large number of successful builds. Similarly, OS-specific faults, which result in fewer build failures than OS-independent faults, might be prevalent and impact the build failure ratio. Therefore, this and the next research question study the relationships between build failures and REs (RQ3), respectively OSes (RQ4).

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" [52]. Each package version is immutable: any change to a package creates a new version. Thus, once a package version starts to fail for a given RE on a given OS, it will keep on failing on future REs on that OS, unless the failure is due to a broken RE fixed in a later version. This RQ studies how often RE versions break builds and whether a failing build can recover or will keep on failing.

Approach. Given a package and an OS, we generate RE build vectors (see Section 3.3), which encode chronological sequences of build results across REs in which "0" and "1" represent failing and successful builds, respectively. We ignore RE versions with missing builds, e.g., the Perl version build vector for Cygwin in Figure 1 would be $[1, 1, 1]$. As explained earlier, we use majority voting to encode build results into zero or one when, for a configuration of an RE and an OS, some builds fail while others succeed (red/green cells in Figure 1). If 50% or more of the builds fail for a given configuration, we put 0 in the corresponding Perl version build vector, otherwise 1.

Then, we analyze 4 possible patterns in the Perl version build vectors, which Table 3 summarizes and Figure 1 illustrates. The pattern for OpenBSD is 0-0, because version 0.004002 of the `List-Objects-Types` package started and ended up failing on multiple REs with some successful builds in between (Perl versions 5.14.4, 5.16.0, and 5.16.2). On the other hand, the pattern for Cygwin is 1-1, while that for Linux is 0-1. The figure does not contain an instance of 1-0.

Findings. *Builds succeed across all Perl versions for 77% of the RE build vectors and fail across all Perl versions for 6%. Build results fluctuate for the*

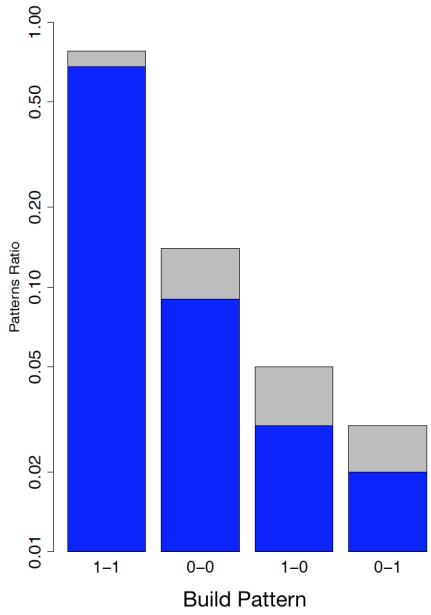

Fig. 9 Percentages of the 4 patterns for Linux. Blue and gray bars represent occurrences of the pure and noisy patterns, respectively.

remaining 17%. Figure 9 shows the percentages of RE build vectors matching each pattern for Linux. Blue bars show the percentages of "pure" matches, i.e., matches that do not include the optional parts (between parentheses) of the patterns in Table 3. Gray bars show the percentages of "noisy" matches.

Our observations provide evidence for Lehman's 7^{th} law for 12% of the build vectors (3% noisy for 1-1, 8% pure for 0-1, and 1% noisy for 0-1) because changes to newer Perl versions fixed build failures occurring in earlier versions. These changes include APIs removed in one Perl version and reintroduced in a following version as well as changes in the behavior of a Perl API. For example, in version 0.05 of `Any-Template-ProcessDir`, Linux follows the 1-1 pattern: although the build fails for Perl version 5.13, it succeeds again from Perl version 5.14 on.

Our observations also show that 11% of the RE build vectors follow the 0-0 and 1-0 patterns: builds that always fail after some particular version of Perl. We explain these observations by counting the number of trailing zeros in the vectors as a measure of the time (in terms of Perl versions) during which builds failed for a given OS. We normalized these numbers by the total number of Perl versions on which builds were made. We find that builds fail for the shortest amounts of time for FreeBSD, Linux, and OpenBSD (with trailing build failures accounting for 20% of the builds) while 50% of all builds fail for Cygwin and 33% for Darwin, GNU kFreeBSD, Solaris and Windows.

RQ3: For 77% and 6% of the RE build vectors, builds consistently succeed or fail, respectively, across all Perl versions. In other words, only for 17% of the vectors, build results provide inconsistent information due to REs.

RQ4: How do build failures relate to OSes?

Motivation. The number of builds across REs and OSes is not homogeneously distributed, which gives some failures more weight than others. No previous work studied the relationship between OSes and build results, except for a brief mention of different build environments in Travis CI by Beller et al. [4], and, more recently, the work of Gallaba et al. [41]. This RQ analyzes whether build failures are specific to certain OSes. If an OS is less popular than others among developers, it might be used less often to build package versions and the corresponding REs on that OS may miss some APIs, both of which are likely causes of build failures. Yet, due to the effects of build inflation, these failures would weigh little in the build results for all OSes.

Approach. We generate OS build vectors, which represent the build results of package version across all OSes for a given RE version and in which, again, zero indicates a build failure and one a successful build (using majority vote). In contrast to RQ3, RQ4 does not study chronological differences in build results, but the consistency of the build failures across OSes. Consistent build failures are more likely to appear for REs with builds on a small number of OSes than on 10 OSes, hence we perform our analysis in function of the vector lengths, from 3 to 10. We group the vectors into separate sets C_i:

$$\begin{bmatrix} B = \{\text{OS build vectors across all package versions}\} \\ C_i = \{b \in B \mid |b| = i\}, \forall i : 3 \leq i \leq 10 \end{bmatrix} \tag{1}$$

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

For a vector length i from 3 to 10, C_i is the union of the vectors (1) that consistently failed or succeeded (C'_i) and in which a majority (C_i^M) and a minority (C_i^m) of OSes have build failures.

Table 4 shows the percentages of vectors with inconsistent builds as well as how often these are caused by a minority of build failures. For a given vector, a minority of m OSes with build failures counts as $\frac{1}{m}$ for each of the OSes:

Table 4 Percentages of OS build vectors in C_i that fail inconsistently and percentages of these vectors for which a minority of OSes is failing (C_i^m). The latter percentages are 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'')	Win %	Linux %	Darwin %	Solaris %	FreeBSD %	OpenBSD %	NetBSD %	Cygwin %	kFreeBSD %	Midnight %
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 more OSes fail together, the lower the weight of the failures, because such failures are less tied to one specific OS.

Findings. *A median of 13.5% of OS build vectors fails inconsistently.* Table 4 shows that the percentages of inconsistent build failures varies from 9% ($N = 10$) to 16% ($N = 6$ or $N = 7$). Since a median of 86.5% of the OS build vectors have no build failures or have build failures for all OSes, this indicates repetitive build results across OSes and, hence, build inflation. Within the 86.5% of consistent OS build vectors, the 11.7% of build failures that occur across all OSes are due to missing features or incorrect logic.

Within the 13.5% inconsistent build vectors, a median of 71% have only a minority of failing OSes: Windows (30%), Linux (7%), and Solaris (7%). Windows is the source of most of the minority inconsistencies, which is likely due to its lower popularity amongst Perl developers, as shown in RQ2. Linux is responsible for more minority inconsistencies when built with a small number of other OSes (small N, e.g., $N = 3$). Conversely, Cygwin and Windows are responsible for more minority inconsistencies in larger sets of OSes (large N, e.g., $N = 10$). GNU kFreeBSD and MidnightBSD are the least inconsistent OSes. The fact that a median of 71% of the inconsistently failing build vectors has a minority of failing OSes (instead of most of the OSes failing), also indicates build inflation.

Furthermore, build failures occurring consistently on most of the OSes, would not require further builds on other OSes, once identified on one OS for a particular package version and RE version. Instead, build failures with inconsistent occurrences across OSes require additional builds to circumscribe their root faults.

RQ4: Only a median of 13.5% of OS build vectors fails inconsistently. A median of 71% of these inconsistent vectors have only a minority of OSes failing.

RQ5: What are the different types of build faults?

Motivation. Identifying different types of build faults and their relationships with REs and OSes can help developers to resolve or even prevent future build failures.

Approach. We used the open coding approach described in Section 3.3 to identify the major types of build faults responsible for the build failures studied in this paper.

Finding. We obtained 6 main types of build faults with 9 subtypes, summarized in Table 5. We give brief definitions of the fault types and a sample of the corresponding build failures.

- The "Dependency" type deals with unfulfilled API dependencies, e.g., missing packages. Typically, these packages do not exist on CPAN but do on the developers' machines. For example:
  - The failure `Can't locate *.pm in @INC` occurs when a Perl module cannot be found in the runtime path `@INC`, because either the package was not installed or the runtime path does not contain the location of the module.
  - The failure `Error while loading shared libraries: ?: cannot open shared object file` refers to missing OS libraries, such as unavailable C/C++ libraries, DLL, etc.
  - The failure `Makefile: recipe for target 'test_dynamic' failed.` pertains to uninitialized dependencies during the install of a package.
- The "Programming" type relates to uninitialized variables, implicit declaration of functions, typos, incorrect data types, and syntax errors. These faults imply that the code cannot be compiled or executed correctly. For example:
  - "Undefined value", e.g., `Use of uninitialized value $class_ip in concatenation(.) or string`.
  - "Data format", e.g., `Non-ASCII character seen before =encoding. Assuming UTF-8`.
  - "Typos", e.g., `prototype mismatch: 2 args passed, 3 expected`.
- The "Environment" type includes faults that occur when trying to access folders, resources, etc. For example:
  - The "configuration" subtype corresponds to errors in build scripts of packages, e.g., `MAKE failed: No such file or directory`.
  - The "security" subtype covers permission issues, such as `Can't exec ‘‘vim’’: Permission denied`.

Table 5 Percentage of occurrence of fault types in REs with minority failures vs. majority failures.

Fault Type	Subtype	Description	Minority Failure %	Majority Failure %
Dependency	Missing module	Package not installed	35.8	27.8
	Missing library	Library not installed	3.6	0
	Unfulfilled deps.	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	Config. errors, e.g., wrong directories	12.6	6.9
	I/O	I/O errors, e.g., serial port issues	3.3	6.9
	Security	Config. errors, e.g., permission denied	2	0
Test	Test	An automated test fail	6	2.8
	Dep.-related test	Failing test due to missing module(s)	4	1.4
Unclear		Improper error message	2.6	5.6

  - When a build cannot perform I/O at the right time, it often fails with `I/O fault: Could not execute: open3: Resource temporarily unavailable`. Similarly, we also observed build faults related to display or serial port.

- The "OS" type covers faults related to unsupported features in some OSes and behavioral differences among OSes, e.g., `your vendor has not defined POSIX macro VEOF` and `It seems localtime() does not honor$ENVTZ when set in the test script`, both occurring on Windows. We observed that almost all OS-related faults occur in Cygwin, Solaris, and Windows with only a minority of OSes failing (minority vectors). This fault type confirms our findings in RQ4: inconsistently failing builds have only a minority of OSes failing.
- The "Test" type refers to failing automated tests, e.g., `release-pod-syntax.t these tests are for release candidate testing`, which indicates semantically-incorrect behavior of the package.
- The "Unclear" type represents faults for which an improper message appears in error logs.

RQ5: Build faults belong to 6 main groups (divided into 9 subgroups): dependency, programming, environment, OS, test, and unclear.

RQ6: How do build fault types relate to OSes?

Motivation: This question identifies the most common faults resulting in majority failures, i.e., failures across most of the OSes. It compares those to the most common faults resulting in minority failures, thus identifying the reasons for builds to fail on only a few of the OSes. Such faults are difficult to detect because they are drowned in a larger volume of successful builds or builds failing across the majority of OSes.

Approach: We use the minority and majority OS build vectors from RQ4 and the fault types from RQ5 to determine the prevalence of each fault (sub)type across OSes. With majority failures, most of the OSes fail because of the same fault while with minority failures, only a few OSes fail, yielding information about OS-specific fault types.

Findings: *"Dependency" faults are the most common reason of build failures overall, followed by "Programming" faults for majority failures and "Environment"/"OS" faults for minority failures.* Table 5 (last columns) shows the percentages of occurrences of each fault (sub)type between the minority and majority build failures, while Figure 10 displays these numbers. The dependency faults dominate both majority and minority failures, followed closely (for majority builds) by the "Undefined value" subtype. The popularity of programming faults among majority faults is expected because an "Undefined value" or "Typo" cannot be fixed by changing the OS. On the other hand, the "OS" and "Configuration" (sub)types intuitively make sense as the second top fault types for minority builds.

Windows, Cygwin, and (to some extent) NetBSD are minority failing OSes experiencing a variety of faults. Figure 11 shows the distributions of fault types across all OSes. The darker a cell (colors and percentages are relative to each OS), the higher the percentage of faults for a given OS belonging to the fault (sub)type on the left. Our earlier finding in RQ4—Cygwin and Windows are the most minority fault-prone OSes—can be explained by the variety of fault types occurring with these OSes when they experience a minority failure. If we compare this observation with Linux, the "Data format" fault subtype is responsible for 66.7% of minority failures, and "Configuration" for the remaining 33.3%.

RQ6: "missing modules" (a "dependency" fault subtype) is the most common reason of failures overall, with majority failures also commonly caused by "programming" faults, and minority failures by "environment" and "OS" faults. The OSes experiencing many minority failures do so because of a wide variety of fault types.

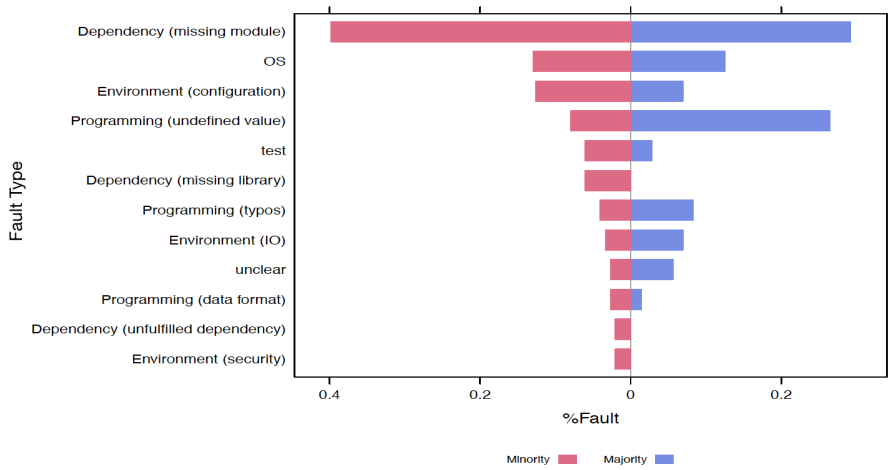

Fig. 10 Distribution of fault types in OS build vectors with minority failures vs. majority failures.

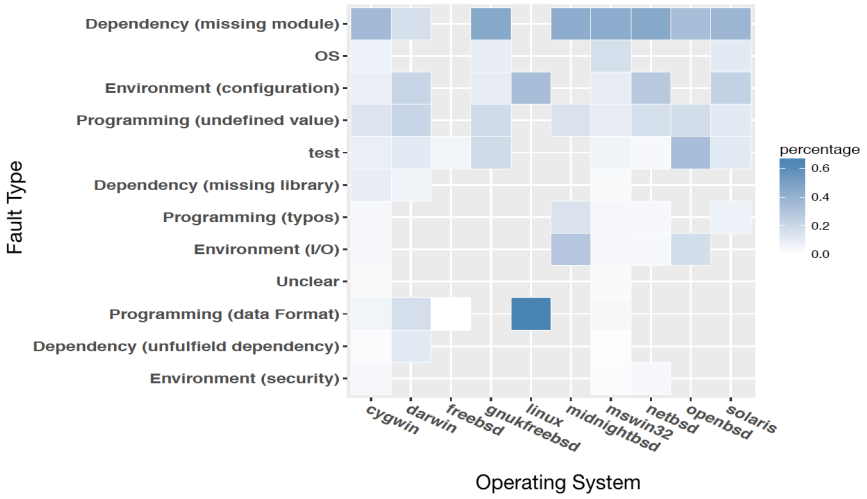

Fig. 11 Distribution of fault categories across all OSes in the minority dataset.

RQ7: How do build fault types relate to Perl versions?

Motivation. This RQ studies failures in different RE versions to find out which failures are temporary or permanent. A temporary failure only occurs for some of the RE versions and is fixed in the more recent versions. A permanent failure occurs in all future RE versions after a given version. We find additional evidence of build inflation by comparing temporary/permanent fault types between minority and majority failures, since permanent failures (by def-

inition) are predictable. Any build on newer RE versions would only provide redundant build results.

Approach. We leverage the 4 patterns of RQ3 to determine whether a fault type is temporary or permanent. We use regular expressions to identify matching error messages in the 791 manually analyzed build failures of RQ5 (333 minority failures and 458 majority failures) across all RE versions: for each most recent failure in a given cell of Figure 1, we identify all its occurrences within its column (RE vector). We then aggregate all non-dependency failures into one group and compare this group to dependency build failures, since RQ6 shows that failures due to API dependencies are significantly more numerous than others.

For example, we search the failure `Can't locate *.pm in @INC` of the package `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 analyze the obtained Perl RE vectors to understand the evolution of the build failures according to the patterns of RQ3.

Findings. *Dependency faults are more difficult to resolve for majority failures than for minority failures.* Figures 12 and 13 compare the distributions of build failures for the 4 patterns of RQ3 between dependency and non-dependency faults in majority and minority builds, respectively. While 72% of the dependency faults of majority failures in Figure 12 follow the 0-0 pattern, only 51% follow this pattern for minority failures in Figure 13. In contrast, the number of 1-1 matches doubles from 14% to 28%. Dependency faults responsible for majority failures are more permanent than those responsible for minority failures.

Non-dependency faults show similar patterns between minority and majority failures. While Figures 12 and 13 show a slight increase in the numbers of 0-1 and 1-0 pattern occurrences for non-dependency faults between majority and minority failures, their orders of magnitude are comparable. This makes sense, since we found, for example, in RQ6 that programming faults dominate the non-dependency faults. Such faults cannot be fixed without changing the source code, which, however, would generate a new package version (with separate build results). Hence, programming faults effectively cannot be fixed within a given package version.

The proportions of occurrences of the 0-0 and 1-1 patterns across all dependency and non-dependency faults are more numerous than those of other patterns: consistently failing or succeeding builds produce repetitive information, i.e., build inflation. The patterns 0-1 and 1-0 (i.e., inconsistent build results) form much smaller proportions of builds.

RQ7: Dependency faults responsible for majority failures are harder to fix than those responsible for minority failures.

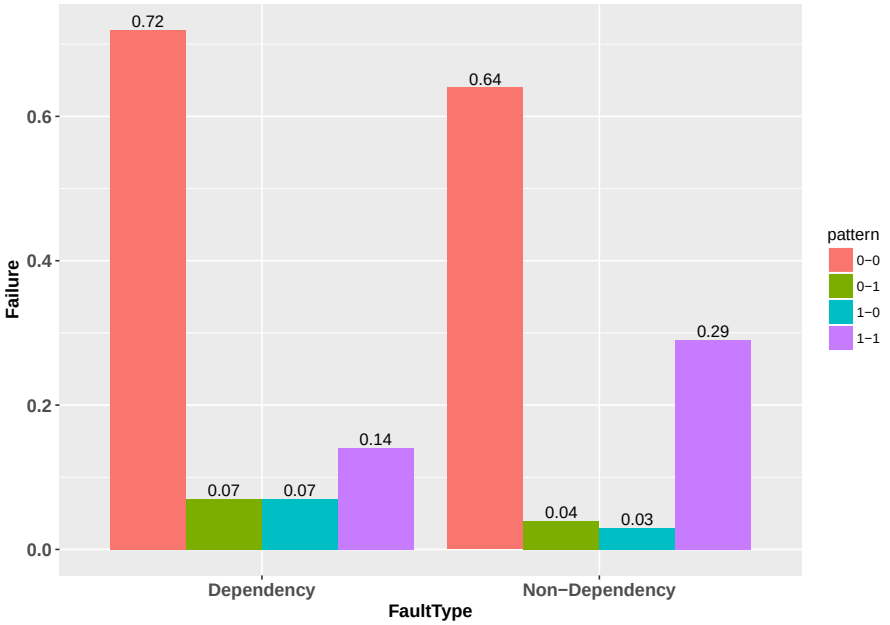

Fig. 12 Distribution of failures due to dependency vs. non-dependency faults in different build patterns when the **majority** (6-10) of builds fail. The y-axis shows the failure ratio and x-axis shows fault types and build patterns.

5 Discussions

By answering RQ1 to RQ7, we now have a better understanding of the frequency of builds and build failures, the distributions of these failures across REs and OSes, and the faults leading to these failures. This section discusses our findings and their impact on practitioners and researchers.

5.1 Implications of Build Inflation for Practitioners

In the introduction, we defined build inflation as the occurrence of a large number of builds for a given commit or release because of the many different kinds of CI build tasks, product variants, and build environments. This paper focused on the latter source of inflation in the form of RE versions and OSes. We report evidence that the higher numbers of builds across environments also render the build outcomes and the interpretation of the build results by developers more complex, e.g., the decreasing ratio of build failures over time in RQ1. In particular, we found the following findings across the seven RQs:

1. Perl development versions are more likely to have build failures than stable Perl versions, yet have less builds.

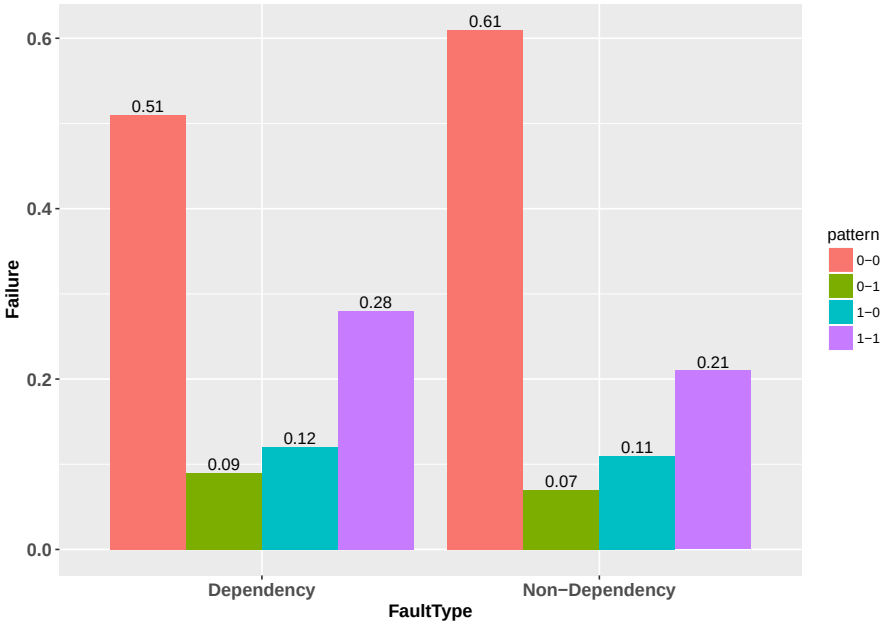

Fig. 13 Distribution of failures due to dependency vs. non-dependency faults in different build patterns when the **minority** (1-3) of builds fail. The y-axis shows the failure ratio and y-axis shows fault types and build patterns.

2. Less popular OSes in the Perl community (Cygwin, Solaris, and Windows) are more likely to have build failures than popular ones (BSDs and Linux) but have less builds.
3. Four out of every five RE vectors yield builds that succeed or fail consistently, while one out of every five vectors yield inconsistently failing builds.
4. For a median of 86.5% of OSes, builds succeed or fail consistently.
5. A median of 71% of inconsistently-failing builds are due to a handful of OSes.
6. The failing OSes (typically the less popular OSes) experience a wide variety of faults.

These findings have implications for CI systems in general and the Perl ecosystem in particular. We now discuss some of these implications, using especially the majority and minority faults identified through our work and available online[4].

5.1.1 CI Systems

Theoretically, every build provides some additional insights about the quality of a software system, regardless of build inflation. Builds can fail on any OS,

[4] `http://www.ptidej.net/downloads/replications/emse19b`

even Linux, which is the number one development OS for the Perl community. If a non-trivial bug has been found and resolved, all builds ideally should be re-run on the bug fix commit/release to assess regressions. However, given the limited resources of companies/communities and the build costs estimated by O'Duinn [11], practically, there are opportunities to reduce the numbers of builds and decrease build inflation by increasing the effectiveness of CI systems.

Our findings suggest that instead of "building each commit/release on all combinations of REs and OSes", CI systems should run a minimal number of builds necessary to have sufficient confidence that (1) the major product variants build and test as expected across (2) the major targeted REs and OSes. For example, CPAN Testers, which relies on machines configured and provided by volunteers, could reduce the number of builds scheduled on BSDs and Linux in favor of machines with Windows. We cannot define explicitly a lower limit of necessary builds and build configurations. However, similar to Occam's razor, CPAN Testers should strive to reduce as much as possible the number of overlapping build configurations, despite the risk of missing unique configurations exposing particular build failures. This risk already exists because build configurations do not represent *all* possible configurations.

Following this idea of a smaller, focused group of build configurations, and similar to test-case selection [53], build schedulers should also reduce the number of builds scheduled across product variants and build configurations, exploiting similarity between variants and configurations in terms of build results. However, the problem of defining and identifying *similar* variants and configurations raises technical and ethical issues. Answering these questions is out-of-the-scope of this paper and is left as future work, e.g.:

- Are two versions of Windows 10 on the same Intel Skylark processor *really* similar?
- Does the data used to determine if two REs/OSes are similar require privileged information?

The criteria to group build configurations are not trivial. First, they should consider the REs, because we observed that Perl development versions have different failure ratios than stable Perl versions. Second, they should consider the OSes, because we observed different failures ratios among Linux and Windows configurations, with different fault types occurring in minority or majority. Third, they should include other factors like HAs, versions of third-party dependencies, etc., which vary per project. CI systems should provide *to developers* a means for aggregating build configurations and weighting the aggregates.

In addition, CI systems should provide better dashboards to highlight and reorganize build failure data to improve their interpretation. For example, they should allow developers to annotate build failures and–or provide feedback to clarify/assign weights to failures. They would thus ease interpretation in comparison to today's systems, which label a commit/release as failing as soon as one of the scheduled builds is failing, no matter whether the failure

happens across all OSes or on only one. They could distinguish build results by the proportion of "unique" failing build configurations, according to the patterns identified in RQ3 and RQ4.

CI systems could also include models predicting the likely outcome of a build [34, 38], incrementally trained on historical build results. Thus, they could perform builds only when necessary. Given the large percentage of builds consistently failing/succeeding across Perl REs and–or OSes, such predictions seem feasible. Using the predictions, they could further optimize builds across product variants and build configurations based on expected costs/benefits. The previous suggestions complement current practices of scheduling builds for groups of commits/releases to make CI scalable to large software projects [10, 15, 33].

5.1.2 Perl Ecosystem

We reported that the majority of faults are related to missing packages in the build configurations, OS-specific problems, or incorrect configuration of directories or daemons, in particular.

Missing packages include "ERROR: no packlist file found: """, "Can't locate .*.pm in @INC", or "No Module::Signature found". OS-specific problems are illustrated by "Can't connect to display 'unix:0': No such file or directory" or "The deprecated ucontext routines require _XOPEN_SOURCE to be defined". Incorrect configuration of directories are "aspell.h: No such file or directory", "Tidyp.h' file not found", or "cannot find include file: "sp.h"". Such faults should be easy to fix by Perl developers by adopting Infrastructure-as-Code (IaC), i.e., by specifying textually all packages, versions, and configuration files needed in build configurations [8]. Dedicated IaC programming languages like Ansible, Chef, or Puppet, allow specifying such information and instantiating the requested build configurations automatically. If CPAN Testers would use IaC, similarly to Travis CI, for example, a major part of the faults could be avoided.

Other faults are due to the Perl programming language itself. Such faults include "implicit declaration of function 'getdcwd"', "Useless use of numeric gt (>) in void context", "Can't bless non-reference value", "Can't call method "get_request" on an undefined value", "Can't use an undefined value as a symbol reference.", "Use of uninitialized value $ExtUtils::F77::Runtime in concatenation (.) or string", and many others. A majority of these faults is related to the nature of the Perl language: a dynamically-typed scripting language. As such, our findings relate to earlier studies on the impact of dynamically-typed languages on software quality, in which faults exist that could be caught by some static analysis (e.g., a compiler). For example, Gao et al. studied the relation between dynamic/static typing and bug-proneness in JavaScript and reported that dynamic languages let significantly more bugs slip through to production [54].

To avoid such faults, the Perl community (or a subset thereof) could decide to follow the example of JavaScript and TypeScript by implementing a

superset of the Perl programming language with mandatory or optional types annotations. They could also use dedicated static analyses to identify such faults, as further discussed in the next subsection. We did not observe other faults that would require other changes to the Perl language or to its use by developers.

5.2 Implications of Build Inflation for Researchers

Our findings lead us to argue that research on build results (including failures) must consider build configurations, the programming language version (and that of its packages), and the nature of the programming language itself.

5.2.1 Build Configurations

Not every build result has the same value and, similar to Simpson's Paradox, aggregating all build results together yields incorrect trends and interpretations. One example is the illusion of decreasing build failure ratio in RQ1, which is mostly due to a proportionally higher number of repeating build successes on the main build configurations, while faults on the other (minority) build configurations are overlooked.

We performed a small quantitative study in which we built explanatory random-forest models[5] with build failure as the dependent variable and with OSes and REs as independent variables. We built the models at the granularity of packages. Out of the 12,584 packages considered in this paper, we selected the 3,949 packages with more than 200 builds and between 20% and 80% of failures: enough data to perform cross-validation. We grouped related packages based on the first parts of their names, e.g., Acme, Net, Yahoo, etc. to avoid having one model per package and over-fitting. We thus built 677 models for 677 groups of packages.

We thus obtained a set of random forest models able to classify a given build as either successful or failing [55]. We used 10-fold cross-validation to evaluate the stability of the models and calculated the area under the ROC curve (AUC) for comparison against random guessing (AUC>>0.5). We also calculate the percentages of build failures classified correctly as build failures (true positive recall) and of successful builds classified as such (true negative recall). The higher these percentages, the better OSes and REs explain build failures.

A median of 88% of successful builds and 80% of failing builds are correctly classified as such using only OSes and REs. Furthermore, Figure 14 shows that most of the models have an AUC value greater than 0.8: they perform substantially better than a random guess. We can also report

[5] The models are not useful to predict build failures in practice because they only include OSes and REs and ignore other factors. However, they are useful to validate the extent to which OSes and REs *alone* explain build failures, i.e., to validate the strength of the link between build configurations and build failures.

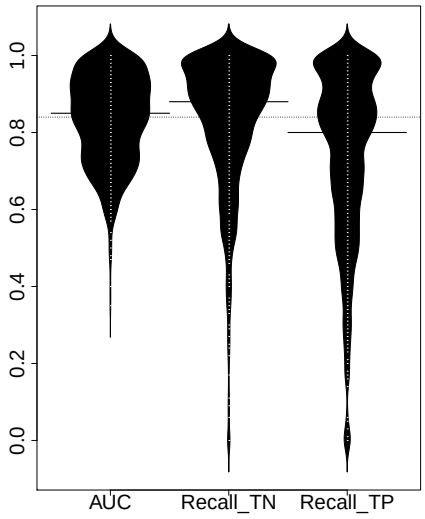

Fig. 14 Beanplot showing the distributions of AUC, true negative recall and true positive recall across all packages. The horizontal lines show median values, while the black shape shows the density of the distributions.

that OSes have a higher explanatory power than REs, based on the outcome of the AUCRF algorithm [56], which implements a backward-removal process according to the primary ranking of the variables. Hence, we confirm that build configurations are an important variable explaining build failures and suggest that researchers study and propose means to identify and resolve (for example through IaC) configuration faults early.

5.2.2 Programming Languages

Developers could expect that a programming language allows straightforward development and sharing of packages, in the sense that, if the package is working on the developers' machines and tested successfully on their CI systems, then they should as well on their users' machines. However, a programming language is more than just its syntax, grammar, and semantics. It also includes its RE (compiler, interpreter, and other packages) and the interactions among REs and the OSes. Hence, a package working and successfully tested on one machine could easily fail on another machine with different packages and OS.

This situation arises from the differences between programming-in-the-small and programming-in-the-large [57]. When programming-in-the-small, a developer (or a small group of developers) develops a package (or program) for one given combination of programming language, RE, and OS. When programming-in-the-large, a large group of developers develops and maintains

a package over an extended period of time, which must work for many different versions of the language, REs, and OSes.

These differences have always existed with programming languages, since the advent of the first computers, and gave rise to the first modern programming languages, in particular Fortran. Java faces this problem as well, and Sun Microsystems and Oracle have offered different versions of Java for embedded systems (Java ME), regular computers (Java SE), and enterprise servers (Java EE). Perl is no different to Java, which explains why some packages may fail in different combinations of REs and OSes. In addition, as explained in the next subsubsection, the dynamic nature of Perl compounds these differences.

The use of different programming models could alleviate this situation, but cannot entirely prevent it. Component-based software engineering or containers could help (1) by making explicit the language versions, packages, and OSes required by a package and (2) by providing these versions, packages, and OSes independently of the underlying machine. However, components and containers also have problems and even recent technologies currently have limitations when it comes to reproducibility, e.g., [58].

5.2.3 Nature of Programming Languages

Other findings discussed in the previous subsection show that the nature of the Perl language is the root cause of many faults, typically faults due to typing errors at runtime. Thus, researchers could also adapt existing and–or propose novel static analyses to Perl to identify these faults early on. There have been long lines of research on dynamically-typed programming languages, for example Smalltalk [59, 60] or JavaScript [61, 62]. Perl could also benefit from such work. Besides researching and implementing static analyses, we can only recommend Perl developers to invest in code reviews and testing (unit tests, regression tests) to identify such faults as early as possible and before production. We cannot recommend other changes to developers' practices, because these faults are intrinsic to the nature of Perl.

6 Threats to Validity

Regarding threats to external validity, this study focused on build data of CPAN Testers, related to packages implemented in Perl. Despite the large number of builds, OSes and REs, we cannot generalize our observations and answers to other programming languages. Yet, the various similarities that we discussed between CPAN Testers and OpenStack Zuul [6] or Mozilla Treeherder [7], as well as commonalities with other studies [3], encourage replication studies on those CI systems.

Regarding construct validity, we fetched the build result data from the centralized CPAN Testers build archive, which is used as the basis for all build reports for the language. As explained in Section 3.3, we performed various filtering steps that could impact the outcome of our study. We explicitly listed

all selection criteria used to replicate our findings. Such filtering is typical for build-result analyses, for example in the recent work on Travis CI [4].

Regarding internal validity, we observed that multiple builds may occur for a single configuration of OSes and REs. These builds would typically exercise a package version on different variants of an OS/RE and even on different HAs. We ignored HAs and other factors, and, depending on the RQ, we considered all builds or subsets thereof for a particular configuration of OSes and REs. Future work should analyze in more detail the relation between HAs and OSes/REs.

Build failures may also be due to flaky tests, i.e., tests that fail inconsistently across different builds because of asynchronous calls, multi-threading, or test-order dependencies. Existing work on flaky tests [63–66] focused on the causes/detection of these flaky tests. They showed that most flaky tests are environment-independent [65], i.e., related to majority-build failures. Future work should analyze to what extent flaky tests impact build inflation.

Finally, we performed our study in a distributed build environment (CPAN Testers) at the release-level, compared to previous commit-level analyses [3, 4, 35, 37–40]. Future work should replicate our study at the commit-level.

7 Conclusion

"Did the build fail and, if so, why?" is a seemingly simple question that developers ask on a daily basis. It has become much harder to answer in recent years due to build inflation. Instead of a one-to-one mapping between commits/releases and builds, modern CI systems perform multiple builds per commit/release because of (1) the different tasks that CI systems perfom, (2) the different configurations needed to be built, and (3) the different REs/OSes, which are the focus of this paper. The term "inflation" implies that not all those builds are equally useful: certain build failures will be over-emphasized, while others will be hidden.

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

- when aggregating all builds, the number of builds for a given package version can see up to 10-fold increases, while the build failure ratio seemingly decreases substantially (RQ1);
- a given CPAN package version is built on dozens of Perl versions and OSes, many of which are not stable, equally popular or supported (RQ2);
- many repetitive builds with predictible outcome are performed across these different environments, adding nuance to the results of RQ1 (RQ2);
- the builds of a working release may fail due to changes in the installed Perl version, with only a small chance for recovery (RQ3);
- some OSes, especially the less common ones amongst developers and users, are more prone to failing their builds (RQ4);

– the most common build fault categories are dependency, programming, environment, OS and test faults (RQ5);
– while missing modules (dependency faults) are the main reason of failures in both minority and majority failures, majority failures are also likely to occur due to programming faults (OS-independent), and minority failures due to environment faults (OS-specific) (RQ6);
– the type and impact of dependency faults differ between minority and majority failures, while we did not find such difference for non-dependency faults (RQ7).

While replication studies on other CI systems and programming languages are necessary, we argue for more "clever" CI systems that (1) optimize the number and type of build tasks and configurations that are scheduled and (2) offer means to meaningfully aggregate build results in order to reduce the overload of build inflation for developers. Researchers could play an important role by developing and evaluating scheduling and aggregation approaches. Furthermore, they should consider the build environment as a control variable in future build-result studies.

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

References

1. P. Duvall, S. M. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.
2. 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.
3. 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.
4. 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.
5. M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, p. 122, 2006.
6. "Openstack zuul ci dashboard," http://zuul.openstack.org.
7. "treeherder," https://treeherder.mozilla.org/#/jobs?repo=mozilla-inbound, accessed: 2017-09-20.
8. J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.
9. L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*, 1st ed. Addison-Wesley Professional, 2015.
10. J. Micco, "Continuous integration at google scale," https://www.slideshare.net/JohnMicco1/2016-0425-continuous-integration-at-google-scale, April 2016.
11. J. O'Duinn, "The financial cost of a checkin (part 2)," https://oduinn.com/2013/12/13/the-financial-cost-of-a-checkin-part-2/, December 2013.

12. "CPAN comprehensive perl archive network," http://www.cpan.org, accessed: 2015-12-22.
13. 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.
14. "CPAN testers," http://www.cpantesters.org, accessed: 2015-12-22.
15. T. Carrez, "Openstack testing automation," February 2014.
16. S. I. Feldman, "Makea program for maintaining computer programs," *Software: Practice and experience*, vol. 9, no. 4, pp. 255–265, 1979.
17. Wikipedia, "List of build automation software," November 2018.
18. G. Booch, *Object-oriented Analysis and Design with Applications (2Nd Ed.)*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1994.
19. Q. Tu and M. W. Godfrey, "The build-time software architecture view," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ser. ICSM '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 398–. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2001.972753
20. P. Kruchten, "The 4+1 view model of architecture," *IEEE Softw.*, vol. 12, no. 6, pp. 42–50, Nov. 1995. [Online]. Available: https://doi.org/10.1109/52.469759
21. 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.
22. 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.
23. S. McIntosh, B. Adams, Y. Kamei, T. Nguyen, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, Waikiki, Honolulu, Hawaii, May 2011, pp. 141–150.
24. S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan, "A large-scale empirical study of the relationship between build technology and build maintenance," *Empirical Software Engineering*, vol. 20, no. 6, pp. 1587–1633, 2015.
25. 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.
26. C. Macho, S. McIntosh, and M. Pinzger, "Automatically Repairing Dependency-Related Build Breakage," in *Intl. Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2018.
27. F. Hassan and X. Wang, "Hirebuild: An automatic approach to history-driven repair of build scripts," in *40th Intl. Conference on Software Engineering (ICSE)*, 2018, pp. 1078–1089.
28. 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.
29. 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.
30. A. Miller, "A hundred days of continuous integration," in *Agile, 2008. AGILE'08. Conference*. IEEE, 2008, pp. 289–293.
31. 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.
32. 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.
33. C. Ziftci and J. Reardon, "Who broke the build?: Automatically identifying changes that induce test failures in continuous integration at google scale," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ser. ICSE-SEIP '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 113–122. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2017.13

34. 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.
35. 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.
36. 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.
37. 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.
38. 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.
39. C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. D. Penta, and S. Panichella, "A tale of ci build failures: An open source and a financial organization perspective," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2017, pp. 183–193.
40. Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, "The impact of continuous integration on other software development practices: A large-scale empirical study," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 60–71. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155575
41. K. Gallaba, C. Macho, M. Pinzger, and S. McIntosh, "Noise and heterogeneity in historical build data: An empirical study of travis ci," in *33rd ACM/IEEE Intl. Conference on Automated Software Engineering (ASE)*, 2018, pp. 87–97.
42. 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.
43. R. O. Rogers, "Scaling continuous integration," in *International Conference on Extreme Programming and Agile Processes in Software Engineering*. Springer, 2004, pp. 68–76.
44. "metacpan-api," https://github.com/metacpan/metacpan-api, accessed: 2016-12-07.
45. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.
46. J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement," *Sociological Methods & Research*, vol. 42, no. 3, pp. 294–320, 2013. [Online]. Available: https://doi.org/10.1177/0049124113500475
47. S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 84–94. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155577
48. "What happens when you push - 2012 edition," https://atlee.ca/blog/posts/blog20120113what-happens-when-you-push-2012-edition.html, accessed: 2017-03-07.
49. 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.
50. 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.
51. cpan@perl.org, "PerlSource versions and release date," accessed: 2016-11-01. [Online]. Available: http://www.cpan.org/src/
52. M. M. Lehman, "Laws of software evolution revisited," in *European Workshop on Software Process Technology*. Springer, 1996, pp. 108–124.

53. S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1002/stv.430
54. Z. Gao, C. Bird, and E. T. Barr, "To type or not to type: Quantifying detectable bugs in javascript," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 758–769. [Online]. Available: https://doi.org/10.1109/ICSE.2017.75
55. 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.
56. 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.
57. F. DeRemer and H. Kron, "Programming-in-the large versus programming-in-the-small," in *Proceedings of the International Conference on Reliable Software*. New York, NY, USA: ACM, 1975, pp. 114–121. [Online]. Available: http://doi.acm.org/10.1145/800027.808431
58. T. Glatard, L. B. Lewis, R. Ferreira da Silva, R. Adalat, N. Beck, C. Lepage, P. Rioux, M.-E. Rousseau, T. Sherif, E. Deelman, N. Khalili-Mahani, and A. C. Evans, "Reproducibility of neuroimaging analyses across operating systems," *Frontiers in Neuroinformatics*, vol. 9, p. 12, 2015. [Online]. Available: https://www.frontiersin.org/article/10.3389/fninf.2015.00012
59. G. Bracha and D. Griswold, "Strongtalk: Typechecking smalltalk in a production environment," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93. New York, NY, USA: ACM, 1993, pp. 215–230. [Online]. Available: http://doi.acm.org/10.1145/165854.165893
60. E. Allende, J. Fabry, R. Garcia, and E. Tanter, "Confined gradual typing," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 251–270. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660222
61. C. Anderson, P. Giannini, and S. Drossopoulou, "Towards type inference for javascript," in *Proceedings of the 19th European Conference on Object-Oriented Programming*, ser. ECOOP'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 428–452. [Online]. Available: http://dx.doi.org/10.1007/11531142_19
62. A. Chaudhuri, P. Vekris, S. Goldman, M. Roch, and G. Levi, "Fast and precise type checking for javascript," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 48:1–48:30, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133872
63. J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *40th Intl. Conference on Software Engineering (ICSE)*, 2018, pp. 433–444. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180164
64. A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: A study of java projects using continuous integration," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 821–830. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106288
65. Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *22nd ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 643–653.
66. F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2017, pp. 1–12.