Do Not Trust Build Results at Face Value – An Empirical Study of 30 Million CPAN Builds

Mahdis Zolfagharinia
MCIS, Polytechnique Montréal
Québec, Canada
Email: mahdis.zolfagharinia@polymtl.ca

Bram Adams
MCIS, Polytechnique Montréal
Québec, Canada
Email: bram.adams@polymtl.ca

Yann-Gaël Guéhéneuc
Ptidej Team, Polytechnique Montréal
Québec, Canada
Email: yann-gael.gueheneuc@polymtl.ca

***Abstract*—Continuous Integration (CI) is a cornerstone of modern quality assurance, providing on-demand builds (compilation and tests) of code changes or software releases. Despite the myriad of CI tools and frameworks, the basic activity of interpreting build results is not straightforward, due to not only the number of builds being performed but also, and especially, due to the phenomenon of build inflation, according to which one code change can be built on dozens of different operating systems, run-time environments and hardware architectures. As existing work mostly ignored this inflation, this paper performs a large-scale empirical study of the impact of OS and run-time environment on build failures on 30 million builds of the CPAN ecosystem's CI environment. We observe the evolution of build failures over time, and investigate the impact of OSes and environments on build failures. We show that distributions may fail differently on different OSes and environments and, thus, that the results of CI require careful filtering and selection to identify reliable failure data.**

I. Introduction

To enable swift detection of bugs and other quality issues, *continuous integration* (CI) has become a main staple in the quality assurance tool box of companies. A CI environment such as Jenkins kicks off a build multiple times a day, either for each patch currently under review, each new commit entering the version control system or at set times during the day (e.g., nightly build), with the aim of notifying developers as soon as possible of breakage [1], [2]. Such a CI build basically combines build and test scripts to run compilers and other tools in the right order, then test the compiled product [3], [4]. -

This paper focuses on studying the impact of this "build inflation", for instance there are many build results across all OSes/environments, which are not necessarily uniform, so each additional build only has diminishing returns. Figure 3 shows this decreasing number of failure, while number of builds increase. Therefore testing in some OS/environment,e.g., Darwin in figure 1, would be more valuable than others,e.g., Freebsd. To perform our analysis of build inflation and its resulting bias, we empirically study 30 million builds of the Perl Comprehensive Perl Archive Network (CPAN)'s [5] CI environment, made between 2011 and 2016. These cover more than 12,000 CPAN packages ("distributions"), 27 OSes and 103 environments to answer the following four questions:

RQ1: How do build failures evolve across time?

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

RQ3: To what extent do environments impact build failures?

RQ4: To what extent do OSes impact build failures?

Our findings can help researchers as this inflation could be responsible for bias in build results. The results can also help practitioners to prioritize the best OS/perl version to test on, in terms of expected value. As a community interested in CI (this year's MSR mining challenge is about CI and builds[1]), we lack basic knowledge on the breadth and depth of the adoption of CI by open-source projects. We need answers to questions such as: does CI advantages surpass its disadvantages? Do developers use CI? We show in the following that CI causes an inflation in the number of builds and build failures, which hide the reality of builds in noise. We also show that unnecessary builds could be avoided by investigating the impact of environments and OSes on build failures. Our observations provide empirical evidence of the bias introduced by build inflation and provide insights on how to deal with this inflation. This, we provide the larger quantitative observational study to date on build failures and build inflation. The results of our study form the basis of future qualitative and *quantitative* studies on builds and CI.

The remainder of the paper is organized as follows: Section II presents important background regarding the CPAN CI environment and major related work. Section III describes our observational study design, while Section IV presents our observations, followed by a discussion of our results in Section V. Section VI focused on threats to validity. Finally, Section VII concludes with insights and future work.

II. Background

A. CPAN

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

[1]http://2017.msrconf.org/#/challenge

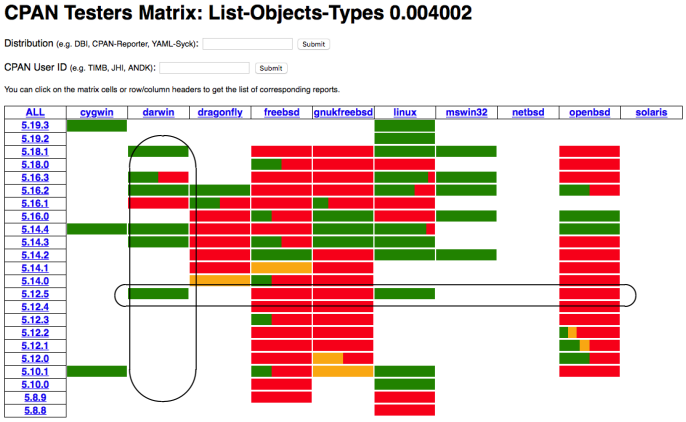

Fig. 1. Example of CPAN build report summary. A vertical ellipse represents an "environment build vector" (RQ3), while a horizontal ellipse represents an "OS build vector" (RQ4).

one or more modules with their documentation, tests, build and installation scripts. Each distribution can have one or more *versions*. To simplify terminology in the following, we will refer to a distribution of a set of modules as a "dist" and to a distribution version as a "distversion".

Build Reports: CPAN implements its own continuous integration system, which builds and tests any new beta or official version of a dist on a variety of operating systems (e.g., Windows vs. Linux) and run-time environments (e.g., Perl version 5.8 vs. 5.19). If successful, the new distversion can be made available to CPAN users. Unlike TravisCI and Jenkins, but similar to Maven and npm, CPAN's CI does not work at commit-level, but at release-level. Since CPAN depends on a mixture of own build servers and build machines owned and hosted by volunteering CPAN members, distversions are not guaranteed to be built and tested on every OS and environment version (cf. white cells in Figure 1).

As Perl is interpreted, build scripts typically process or transform scripts and data instead of mere compilation. The test scripts then allow to verify that the modules in the dist are working correctly on a given OS and environment. For each build (we will refer to the execution of build and test scripts for a given OS and environment as "build"), CPAN generates a build report and all reports for a given version of a dist are summarized into an overview report, as shown in Figure 1. This Figure shows the summary of build results for version 0.004002 of the "List-Objects-Types" dist for (left column) Perl environments 5.8.8 to 5.19.3, and OSes (top row) CygWin to Solaris. Red cells indicate that all builds for a combination of OS and Perl environment failed, green cells show that all were successful, red/green cells indicate that some builds failed, and orange cells represent unknown results (e.g., build or tests were interrupted).

Rest API: CPAN provides a RESTful API [6] and a Web interface [5] to create complex queries on all publicly available modules and dists in CPAN. Furthermore, the whole history of CPAN and all of its modules and dists is accessible via the GitPAN project[2]. These data sources allow to conveniently access CPAN for analysis, providing access to not only build results, but also module meta-data, such as modules names, versions, dependencies, and other helpful information:

- GUID: a unique global identifier that identifies each dist.
- CSSPATCH: a value (`pat`) or (`unp`) indicating if this module was tested with a patched version of Perl or not.
- CSSPERL: a value (`rel` or `dev`) indicating whether a release or development version of Perl was used to test.

B. Related Work

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

Denny et al. [7] investigated compile errors in short pieces of Java code and how students fixed these errors. They showed that 48% of the builds failed due to compilation errors. Similarly, Dyke [8] assessed the frequency of compile errors by tracking Eclipse IDE usage with novice programmers.

Suvorov et al. [9] studied two popular open-source projects, Linux and KDE, and found that the migration from one build system to another might fail due to missing features. Adams et al. [3], [10] analyzed the changes to the Linux kernel build system and reported that the build system grew and had to frequently be maintained. McIntosh et al. [11] replicated the same study on the ANT build system. We complement these studies with our predictive model of build failures for the build system of CPAN.

Other CI systems similar to CPAN are Jenkins, Hudson, Bamboo, and TeamCity [12]. Stahle and Bosch [13] surveyed CI practices and build failures in CI systems. They observed that test failures during builds are sometimes accepted by developers because developers known that these particular failures will be fixed later [14]. We complement this study that is showing the impact of OSes and environments on build failures within CPAN particular unnamed CI system."

Seo et al. [4] reported on a case study performed at Google, in which they assessed 26.6 million builds in C and Java. They showed that there exist multiple patterns of build failures and focused on compilation problems to conclude that dependencies are the main source of compilation errors, that the time to fix these errors vary widely, and that developers need more, different tools support. We complement this study with a larger-scale study of 68.9 millions builds of CPAN.

Vasilescu et al. [15] conducted an empirical study about CI usage on 246 projects from GitHub. They showed that CI significantly improves productivity of GitHub teams. However, despite providing evidence on the benefits of CI, they do not provide any detailed information about CI usage. We provide evidence supporting Seo et al. [4] on the inflation of the number of builds.

Hilton et al. [16] assessed 34,544 open-source projects from GitHub, 40% of which use CI. They analyzed approximately

[2]https://github.com/gitpan

1.5 million builds from Travis CI to understand how and why developers use CI. They found evidence that CI can help projects to release regularly. Leppanen et al. [17] also reported more frequent releases as a perceived benefits of CI, by interviewing developers from 15 companies. Two other works [18], [19] have performed case studies on the use of CI and found positive impact of CI. This growing usage of CI makes it important to perform a larger study of CI and builds.

The API of an ecosystem is an important factor of development costs [20]. Previous work studied API for various purposes: (1) to recommend relevant API to developers during development and–or during changes, typically when upgrading the dependencies of one module to newer versions of other modules [21], and (2) to migrate API, in particular in the context of services [22]. We refer the interested reader to a previous article summarizing this work [23]. We summarize here only two relevant works related to API and build systems.

Wu et al. [23] followed up on this first study by analyzing changes in 22 releases of the Apache and Eclipse frameworks and their client programs. They observed the kind of API changes in the frameworks impacting the clients and classified API changes and API usages. They suggested analyses and tools to apply on frameworks and their client programs to identify API usages and reduce the impact of API changes.

We provide evidence that CI is a sane practice to identify and mitigate API changes that can occur in modules, OSes, and environments but also that build failures must be analyzed carefully in function of OSes and environments and in function of time."

III. Observational Study Design

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

A. Study Object

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

B. Study Subject

We choose CPAN to study the impact of OS and environments on build failures because CPAN [5] provides the results of the automated builds of all Perl dists and distversions on dozens of OSes and run-time environments, hence it provides a large and rich data set. Moreover, CPAN has a long history although it provides build data only at the release level. We will study data at the commit level, as provided by Travis CI, in future work.

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

C. Study Sample

First, we studied the CPAN build meta-data of 69.8 million builds over the full period of 16 years, between January 2000 and August 2016. We found that most builds were performed between 2011 and 2016. Although, for each OS, builds were performed on different OS versions and architectures, 10 OSes and 13 Perl environments stood out. In particular, each of these 10 OSes had more than one million builds, while each of these 13 environments had more than 800,000 builds. To reduce time and (to some degree) simplify our analysis, we filtered out the other OSes and Perl environments, yielding a data set of 62.8 million builds on 10 OSes and 13 environments (Perl 5.8 to 5.21, excluding 5.09) for a period of about 5.5 years between January 2011 to June 2016. Although we investigate 13 Perl major versions, e.g., 5.8, it includes 103 Perl minor version, e.g., 5.8.8. Therefore, results of a major version includes the results of all minor versions. This data set included dists with only one build, as well as dists with thousands of builds. For example, 13,522 dists have more than 1,000 builds while 967 dists have less than 3 builds. Unfortunately, not all builds have build result data available, and no reliable conclusions can be made for dists with too few builds or too few versions, while at the same time modules with too many builds or versions might not be representative either. For this reason, we filtered out builds without corresponding data, we determined lower and upper thresholds for the number of builds and number of versions, and filtered out modules below or above those thresholds, respectively.

Figure 2 illustrates the distribution of the median number of builds and the number of versions across all CPAN dists in our full data set (the number of dists within each quadrant is shown as well). Black lines show the lower and upper thresholds that we determined for the median number of builds and number of versions for each dist. First of all, by looking at the data distribution, we filtered out CPAN dists with less than 10 build results and less than 5 versions. Then, to determine the upper thresholds for filtering outliers, we used the following formula [24] based on the inter-quartile range: $ut = (uq - lq) * 1.5 + uq$. Here, lq and uq are the 25th and 75th percentiles, while ut is the upper threshold.

Of the nine quadrants shown in Figure 2, only the central one is used in our study. After removing 849,638 builds without data and filtering out the other quadrant data, we obtained a data set of 12,584 CPAN dists with about 30 million

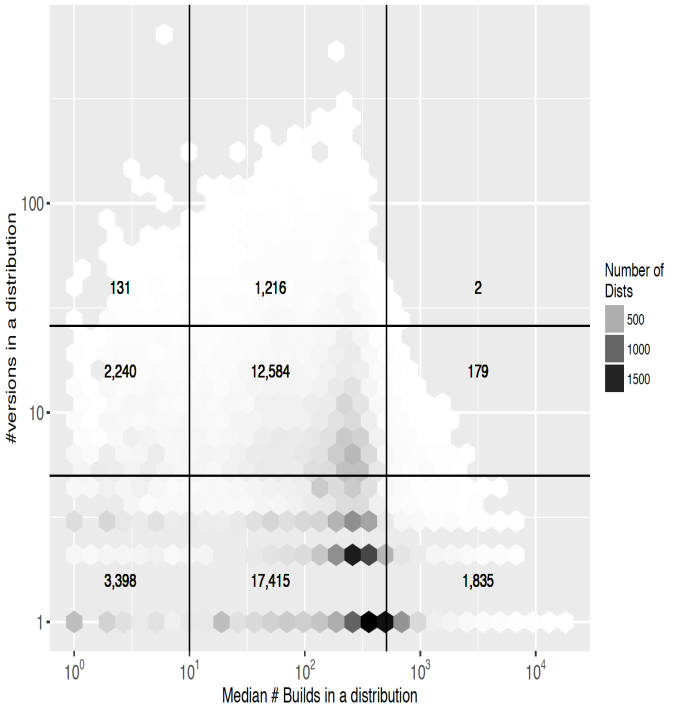

Fig. 2. Distribution of the number of builds and versions across CPAN dists. This hexbin plot summarizes where the majority of the data can be found (darker cells), where a cell represents the number of CPAN dists with a given median number of builds (x-axis) and number of versions (y-axis). The black lines correspond to the thresholds used to filter the data, dividing the data into 9 quadrants. For each quadrant, the number of CPAN dists within it is mentioned on the plot. The central quadrant contains the final data set.

builds. Including other quadrants would increase our data set size, but might introduce noise in the form of outliers.

IV. Observational Study Results

We now present the motivation, approach and results of the four observational research questions, RQ1 to RQ4.

RQ1: How do build failures evolve across time?

Motivation. This initial research question aims at understanding how often builds fail and whether the ratio of failing builds is a constant value or fluctuates across time. So we investigate build inflation in terms of number of builds. The former has been looked at by recent studies, for example Beller et al. [1] found a median of 2.9% of Java builds and a 12.7% of Ruby builds in TravisCI to be failing, while Seo et al. even recorded failure ratios of 37.4% and 29.7% for C++ and Java builds at Google [4]. Unfortunately, apart from these lump numbers, not much more is known about the build failure ratio of projects, for example about how this ratio evolves over time. Furthermore, all existing CI studies have targeted commit-level builds and tests, while CPAN is a package release-level build environment typical for software ecosystems.

Approach. In order to study the ratio of build failure in our build report data set, we consider all UNKNOWN build results (orange cells in Figure 1) as a failure. For each CPAN dist in the data set of 30 million builds, we then compute the ratio of

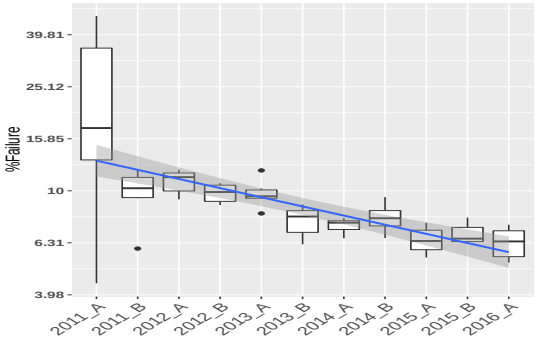

Fig. 3. Distribution of failure ratio in six-month periods. The linear regression line shows the corresponding trend of the ratio over the six studied years.

build failures as $\#buildfailures/\#builds$. We then aggregate this data per period of six months to obtain the evolution of build failure ratio across time. Note that we do not distinguish between OSes and environments in this RQ.

Findings. The median build failure ratio decreases across time from 17.7% in 2011 to 6.3% in 2016.

Figure 3 shows the distribution of failure ratios across all builds of all CPAN dists in the studied period of 6 years. From 2010 to 2014, each year two Perl versions were released, so we investigated the evolution of failure in period of six months. From 2011 to 2013, the median failure ratio in the first half of a year is higher than that of the second half of the year, yet from 2014 on this trend is reversed. However, as the regression line in Figure 3 shows, the overall build failure ratio has a strong decreasing trend between 2011 and 2016, especially when taking into account the logarithmic scale used in the figure. This decreasing ratio might be due to several reasons, for example less builds being performed across time or less releases being made for dists. In the following, we explore these two hypotheses.

Until the first half of 2015, each year more builds are being made, from one (2011) to several (2016) million. Table I shows the number of builds per period of six months. We observe that, even though ever more builds are executed, they seem more successful over time, i.e., there is an inverse correlation between numbers of builds and build failures. It is not clear why the second half of 2015 and first half of 2016 show a decreasing number of builds, however these observations might explain the plateau (instead of decrease) of median values for the rightmost boxplots in Figure 3.

The average number of builds per distversion shows a three-fold increase from 44.7 to 412.9 across time, although there are some fluctuations from the second half of 2014 on (2014-B). To understand if the decreasing build failure ratio is due to a drop in the number of releases being made over time, we counted the number of releases of dists in each six month period. The average number of builds per release in Table I shows an increasing trend, growing from 44.7 in the first six months of 2011 to 501.7 in the first half of 2015 (with a slight dip end of 2014), after which the average ratio

TABLE I
NUMBER OF BUILDS, DISTVERSIONS AND AVERAGE NUMBER OF BUILDS PER DISTVERSION IN EACH PERIOD OF SIX MONTHS BETWEEN JANUARY 2011 AND JUNE 2016.

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

drops, but still remains higher than in 2014. The latter can be explained as follows: although the number of builds dropped from the second half of 2015 on, the number of releases did not drop at the same rate.

Overall, the steady drop in build failure ratio can (at least partially) be explained by a strong increase in number of builds per release, i.e., strong increases in the number of builds and the number of releases per CPAN dist. While an increasing number of releases is typical for today's release engineering strategies [25], the increasing number of builds cannot be explained intuitively. The next research question helps understand this build inflation by considering the impact of different OSes and environments on builds.

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

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

Motivation. Build inflation seems to correspond to differences/similarities in OSes/environments, which can identify clusters of Perl versions/OSes with different failure ratio and value. To explain the decrease of the build failure ratio observed in RQ1, our hypothesis is that a given new distversion is built multiple times in such a way that most of these builds succeed, while only few fail. One major reason for such build inflation is the need to build and test a release on different versions of the OS and Perl environment. Although each OS and Perl environment of course can show deviating behaviour (which is why multiple builds are performed in the first place), they are essentially building and testing the same features. Hence, feature-related bugs would be expected to trigger across all OSes and environments, inflating the number of build failures due to such bugs. OS- or environment-specific problems would only be reported once for the problematic OS or environment. This deviation might not only explain our findings for RQ1, but also lead to bias in build results that needs to be addressed to avoid incorrect conclusions.

Approach. For each build, we extracted build log data about the OSes and environments used during the build, then calculated build failure ratios per OS or environment type. For the reasons outlined in RQ1, we removed builds with unspecified status. Furthermore, as any CPAN community member can volunteer a machine for CPAN builds, a wide variety of hardware architectures and OS/environment versions is being used. To make our analysis feasible, we did not distinguish between the 573 different hardware architectures in our data set, but treated them as equal. The analysis of the impact of hardware architecture is left as future work.

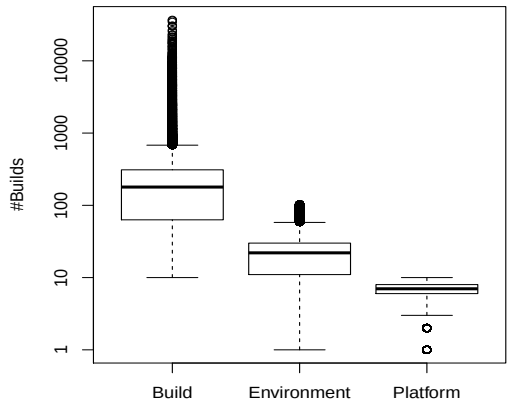

Fig. 4. Distribution of the number of builds per CPAN dist, as well as of the number of OSes and environments on which these dists' builds took place.

Findings. The analyzed CPAN distversions have a median of 179 builds, which take place on a median of 22 environments and 7 OSes. Figure 4 shows the distribution of the number of builds, OSes, and environments across all dists. While the numbers of OSes is more or less stable around 7, the number of Perl environments to test is much higher, while the total number of builds for a dist correlates with the product of both.

Through a manual analysis of the CPAN data, we observed that, when a new version of a dist is released, it should be built and tested on most of the supported OSes and environments to check if the new distversion is backwards compatible with their application programming interface (API) [26]. Conversely, when a new OS or environment becomes available, most of the existing distversions that are not yet deprecated are rebuilt, which explains the inflation of the number of builds over time found in RQ1, but not yet the decrease of the build failure ratio.

Not every environment yields equally reliable build results. Figure 5 illustrates the evolution of build failures from Perl version 5.8 (released in 2002) until 5.21 (2015). Environments (Perl versions) are shown on the X axis, ordered by release date [27], while the Y axis shows the build failure ratio (blue; right axis) and the percentage out of all builds

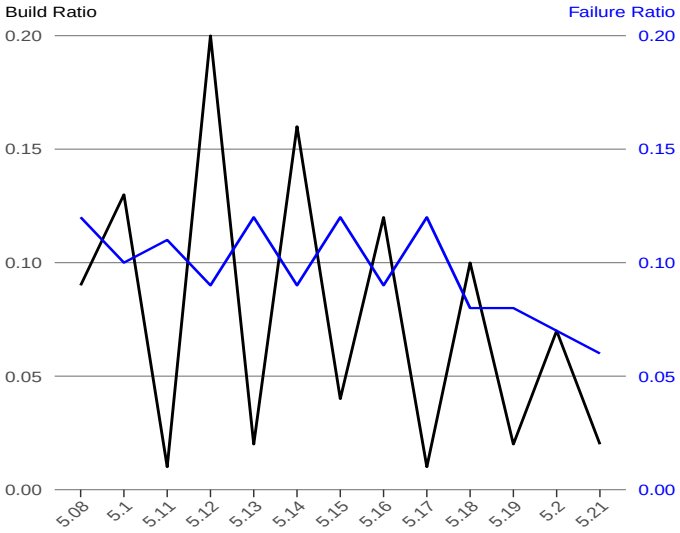

Fig. 5. Distribution of the ratio of all builds performed on a given environment (black y-axis), and the proportion of those builds failing (blue y-axis).

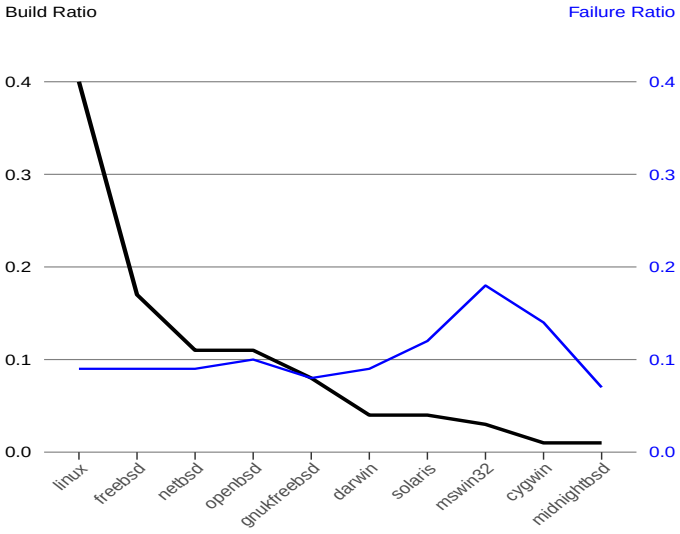

Fig. 6. Distribution of the ratio of all builds performed on a given OS (black y-axis), and the proportion of those builds failing (blue y-axis).

performed on a given Perl version (black; left axis). The jagged trend of the black line (percentage of builds) is surprising, suggesting that odd releases see substantially less builds than even ones.

Closer analysis showed that Perl uses a specific semantic versioning approach [27] where even release numbers, like 5.8 and 5.10, are official production releases (with maintenance releases, such as 5.12.1 and 5.12.2 mainly for bug fixes) and odd release numbers, like 5.11 and 5.13, are development releases. Hence, development releases are used less for builds and have less reliable build results. In particular, versions 5.19 and 5.21 were less failure-prone than their stable successor, while other odd versions were more failure-prone. Furthermore, some builds have been performed on older environments

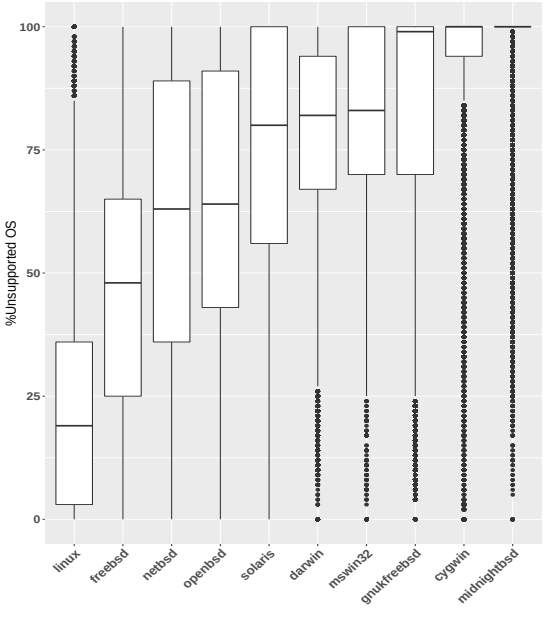

Fig. 7. For a given OS, the distribution across CPAN distversions of the percentage of environments for which **no** builds have been performed.

but there was not enough build data to study these in details.

The most failure-prone build OSes are Windows (%18), Cygwin (%14), and Solaris (%12). Figure 6 shows the percentage of builds and failing builds for different OSes. It shows a clear difference between BSDs/Linux on the one hand and Windows/Cygwin/Solaris on the other hand. The former cluster of OSes has a substantially larger numbers of builds than the latter cluster, while the percentage of failures is lower. Hence, similar to environments, the build results on some OSes are less reliable than others, because they might just indicate a lower popularity of those OSes in terms of builds and development.For some OSes like MidnightBSD or Cygwin, we have very few number of builds. Therefore, we get results based on few amount of data, so it might not be as reliable as the results for Linux with the huge number of builds. Finally, the average and median numbers of OSes on which each CPAN dist is built is 7. Note that this number excludes the OSes that were filtered out in Section III-C due to very low number of builds.

The fact that some OSes are more failure-prone and less popular amongst developers than others can also be observed when measuring the number of times when no build is performed for a distversion on a given combination of Perl version and OS. These cases correspond to the empty cells in Figure 1. The resulting distribution of the percentages of missing builds across CPAN distversions for each OS is shown in Figure 7. We can observe how the most incomplete OSes correspond to those with the most build failures in Figure 6. Therefore, anyone interested in studying build results for CPAN should not blindly trust the results for the least supported OSes, such as MidnightBSD, Cygwin, and Windows, because there are

too few builds to be reliable and meaningful.

> *RQ2: The analyzed distversions are built a median of 179 times on a median of 22 environments and 7 OSes, yet not every OS and environment yields equally reliable build results.*

RQ3: To what extent do environments impact build failures?

Motivation. In RQ2, we analyzed the impact of OSes and environments on the number of builds and build failure ratio to better understand the increase of the number of builds. Although we find empirical evidence of build inflation due to repeated builds on different combinations of OS and environment, we could not yet explain why the ratio of build failures has been *decreasing* over time, as seen in RQ1. We suspect that build failures specific to one environment version will only register as one build failure, while builds of an abandoned distversion will fail on all environments. Similarly, OS-specific bugs will register less build failures than bugs in OS-independent code. Not all such failures are equally valuable to analyze or focus on. Therefore, the next two research questions study the impact of environment- (RQ3) and OS-specific (RQ4) build failures.

Lehman's 7th 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 [28]. As each CPAN distversion is immutable (any changes will generate a new distversion rather than updating an older one), this means that once a distversion starts to fail on a given Perl version, it is expected to keep on failing on future versions, unless the Perl developers are able to fix the Perl APIs. Hence, this RQ aims to understand how often environments break the build and to what degree a failing build can recover again or is doomed to keep on failing.

Approach. For each distversion and OS, we consider the chronological sequence of build results across environments as binary vectors ("environment build vectors"), where a 0 marks a failure and a 1 a successful build. Environment versions with missing builds are ignored. This means that the environment build vector for Cygwin in Figure 1 is [1,1,1], as white cells (missing builds) are ignored. We then analyze the four possible patterns of build failure evolution and study these patterns across our environment build vectors to identify environments that are more failure-prone and tend to keep on failing.

These four patterns are summarized in Table II. For instance, in Figure 1 OpenBSD's failure pattern is 0-0, since version 0.004002 of the List-Objects-Types dist started out and ended up failing on multiple Perl versions, with some successful builds in the middle (Perl versions 5.14.4, 5.16.0, and 5.16.2). Similarly, the pattern for Cygwin is 1-1 (missing builds are ignored), while that for Linux is 0-1, with some fluctuations in the middle.

Since for a given OS and Perl version some builds could be successful while others failed (cells colored partially red and partially green in Figure 1), we used majority vote to summarize these build results into one 0 or 1 value for use in the environment build vector. As soon as 50% or more of the builds for a given OS and Perl version failed, the Perl environment is said to be failure-prone (0 in the vector).

TABLE II
TOTAL PERCENTAGE OF OCCURRENCES OF THE FOUR DIFFERENT BUILD FAILURE EVOLUTION PATTERNS, ACROSS ALL OSES. "PURE" REFERS TO OCCURRENCES OF THE PATTERNS WITHOUT FLUCTUATION (E.G., 111), WHILE "NOISY" REFERS TO OCCURRENCES WITH FLUCTUATION (E.G., 101).

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%

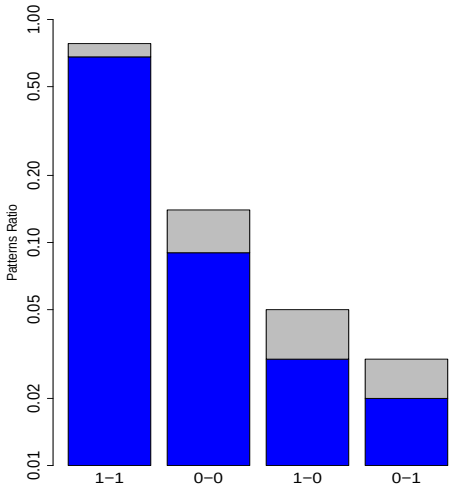

Fig. 8. Percentages of the four build failure evolution patterns for the Linux OS. The blue bars represent occurrences of the pure patterns (no fluctuation) while gray bars are noisy occurrences (including fluctuations).

Findings. For 77% of the environment build vectors, builds succeed across all Perl versions, for 12% the builds eventually succeed towards the most recent Perl versions, and for the remaining 11% the builds eventually fail or never succeeded at all across the Perl versions. Figure 8 shows the percentage of environment build vectors matching each pattern in Linux. Blue bars show the percentage of "pure" matches, i.e., matches that do not include the optional part (between parentheses) of the patterns in Table II. The blue bar for pattern 1-1 represents environments in which builds always succeed. On the other hand, gray bars match the full patterns, including the optional fluctuations from 0 to 1 or 1 to 0, where the APIs of the Perl environments temporarily behaved differently than before.

Our observation provides evidence for Lehman's 7th law of software evolution, in the sense that for 12% of the analyzed tuples (3% noisy for 1-1, 8% pure for 0-1 and 1% noisy for 0-1) changes to newer environment versions have been able to fix previous build failures. These correspond to cases where an

API is removed from a Perl version before being added again, or where the implementation of a given API changed behavior. For example, in dist version "Any-Template-ProcessDir 0.05", Linux follows the 1-1 pattern: although it breaks in Perl version 5.13, it succeeds again from Perl version 5.14 onward.

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

RQ3: 11% of the build vectors where the build starts to fail will never succeed again, while for 12% environment-specific failures occurred that eventually were resolved.

RQ4: To what extent do OSes impact build failures?

Motivation. Although every new run on a new platform can bring new information, the amount of test results across environments and OSes is not homogeneously distributed, which makes some failures to have more weight than others. Except for a brief mention of different build environments in TravisCI [1], related work has not yet studied the impact of OS type on build results. Similar to the idea of build failures being specific to certain environment versions, this RQ analyzes the degree to which build failures are specific to certain OSes. Especially if one OS is less popular than others, it might have seen less testing or some features might not have been ported over, causing test scripts to fail. Such failures would receive less weight in the build results than builds failing consistently across all OSes. Hence, here we are interested in measuring whether certain OSes are indeed more failure-prone than others.Moreover, if build results of specific OSes are similar(BSDs), then we have the choice of testing only one of them for each distversion/Perl version.

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

In contrast to RQ3, RQ4 does not study chronological differences in build results, but rather how consistent builds fail across OSes. Since it is easier to have consistent build failures when a build is only done on 3 or 4 OSes rather than on 10, we split up our analysis across OS build vectors of different lengths, from 3 up to 10. For this, we cluster the vectors into separate sets C_i, as follows:

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

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

In other words, for a given vector length i from 3 to 10, the set C_i is the union of the set of vectors that consistently failed or succeeded (C'_i), the set of vectors where a majority of OSes had a failing build (C_i^M) and the set of vectors where a minority of OSes had a failing build (C_i^m). The former two sets give a high weight to build failures, since most, if not all, of the OSes fail to build, while the latter set consists of build failure anomalies, since few OSes have a different build outcome than the majority of the OSes. (We do not distinguish between sets of OSes (e.g., {CygWin, Windows, Linux} vs. {Darwin, Solaris, NetBSD}), only between lengths of vectors.)

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

Findings. A median of 13.5% of OS build vectors fails inconsistently. Table III shows how the percentage of inconsistently failing build vectors varies from 9 (N=10) to 16 (N=6 or 7). Such build failures are specific to certain OSes. On the other hand, a median of 86.5% of build vectors either had no failed build, or consistently failed on all OSes. The latter build failures are purely feature- or logic-related.

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

TABLE III

PERCENTAGE OF VECTORS IN C_i THAT FAILS INCONSISTENTLY, AS WELL AS THE PERCENTAGE OF THOSE VECTORS FOR WHICH A MINORITY OF OSES IS FAILING (C_i^m). THE LATTER PERCENTAGE IS THEN BROKEN DOWN ACROSS ALL OSES THAT WE STUDIED (I.E., THEY SUM UP TO THE PERCENTAGE IN THE THIRD COLUMN).

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

RQ4: A median of 13.5% of OS build vectors fails inconsistently, with a median of 71% of those having only a minority of OSes failing. Windows (30%) and to some extent Linux/Solaris (7%) figure the most amongst the failing OS minority.

V. DISCUSSION

After answering RQ1 to RQ4, we have a better understanding of the frequency of builds and build failures, and the distribution of these failures across OSes and environments. In particular, we aimed to understand why, despite an inflation of builds across time, the percentage of build failures keeps on decreasing. Here we want to validate to what degree the observations that we made, in particular knowledge about build OS and environment and their impact on build inflation, are able to explain the presence of build failures. Any unexplained failures are due to other factors, such as the source, logic problems, etc.

To do this, we build an explanatory classification model with build failure as dependent variable and as independent variables the OS and environment used for a given build. We use random forest classifiers for our models, which builds an ensemble of decision trees that are used together to classify a given build instance [29]. We use 10-fold cross validation to evaluate the stability of the explanatory models, then calculate the area under the ROC curve (AUC) to understand how much better the models are than random guessing (AUC>>0.5). Furthermore, we calculate what percentage of build failures is classified correctly as a build failure (true positive recall) and what percentage of successful builds is classified as such (true negative recall). The higher these numbers, the better OS and environment knowledge is able to explain build failures.

We build models for different dists. Initially, we chose dists with more than 200 builds, having a failure percentage between 20% to 80%. Thus, from 39,000 dists, we kept 12,584, out of which we kept 3,949 dists, so that we have enough builds to get a rational result from a random forest but we can also apply cross validation.

To have enough data and avoid having thousands of small models, we then grouped related dists into groups based on the first part of their name, e.g., Acme, Net, Yahoo, etc. We thus built 677 explanatory models for 677 groups of dists.

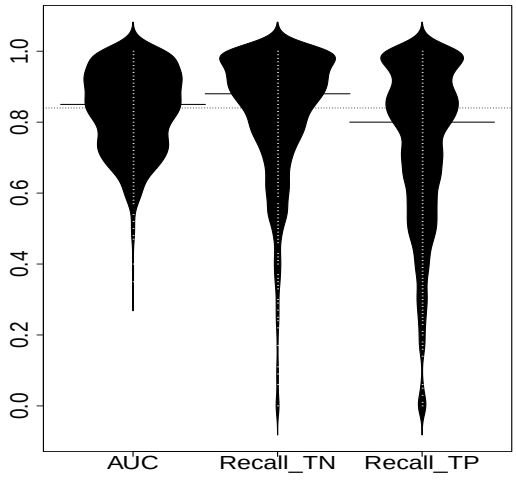

Fig. 9. Beanplot showing the distributions of AUC, true negative recall and true positive recall across all dists. The horizontal lines show median values, while the black shape shows the density of the distributions.

A median of 88% of successful builds and 80% of failing builds are correctly classified as such based only on information about OS and environment. Furthermore, as shown in the beanplots of Figure 9, most of the models have an AUC value higher 80%, which shows that the models clearly contain more knowledge than a random guess model. By using the AUCRF algorithm [30], which implements a backward-removal process according to the primary ranking of the variables, we found that OS has a higher explanatory power than environment for build failure, confirming our earlier findings in RQ4.

Our models are of course not useful in practice to predict build failure, since they only include OS and environment, while ignoring a myriad of other factors, notably related to source code and overall code quality. However, prediction is not the point of the models. Instead, we only use the models to

validate the extent to which knowledge of OS and environment alone are able to explain build failures.

VI. Threats To Validity

This study was performed on build data of the CPAN ecosystem (which is centered on the Perl language). Thus, we cannot necessarily generalize our observations and answers to other ecosystems and programming languages. Yet, the numbers of builds, OSes, and environments provide an interesting basis on which to perform future studies and against which to compare their results.

Regarding construct validity, we fetched the build information from the centralized CPAN build archive, which does not provide the full detailed reports on all errors occurring during builds. We filtered out builds with corresponding failure data in RQ1 to RQ4, hence even through we studied 68.9 millions builds and their results, it is possible that other failures/successes exist.

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

Regarding internal validity, while we studied the impact of OSes and environments on build failures, we did not perform in-depth analyses of the reasons for failures/successes of a given OS version, Perl version or combination thereof, as well as other factors, like architectures, minor/major releases, developer experience, module complexity, etc. Given its scope, manual analysis of build error messages, similar to Seo et al. [4], is also part of future work. From 39,000 distributions in CPAN, we chose those ones which were best fit for the analysis, similarly to recent work on Travis CI [1]. Our results may be different with a different dataset.

VII. Conclusion

Build results have turned out to be a valuable data source for both researchers and practitioners. However, one cannot trust the results at face value due to a phenomenon of build inflation, where individual code changes or package releases yield dozens of builds across different OSes and environments. This inflation artificially blows up the importance of certain build failures, while it hides others. Whereas builds are supposed to give an indication of the quality of a software product, to some degree build results reflect more on portability and platform issues of OSes and environments.

In particular, based on our study of 30 million CPAN builds between 2011 and 2016, researchers and practitioners should be aware that:

- the number of builds for a given software product can see up to 10-fold increases, while the build failure ratio seemingly decreases substantially (RQ1)
- a given software product like a CPAN distversion is built on dozens of environments and OSes, many of which are not stable, popular or commonly supported, and hence are not equally reliable as build platform (RQ2)
- for a given environment, a working software product suddenly can start breaking due to changes in the environment, with only a small chance for recovery (RQ3)
- some OSes are notorious for failing the build (RQ4)

Thus, we conclude that researchers interested in studying build results should analyze and select the results for the main OSes and environments, while ignoring other build results. Moreover, although this observational study is quantitative, it is the larger to date observing build failures and build inflation and will serve as the basis of future other qualitative and quantitative studies on builds and CI.

Future work includes replicating our study at the commit level, using for example the dataset from Travis CI. We will also analyze in more details OSes and environments with multiple versions/architectures. Finally, we want to analyze error messages of build failures as well as study qualitatively these failures.

Acknowledgment

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

REFERENCES

[1] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An analysis of travis ci builds with github," PeerJ Preprints, Tech. Rep., 2016.

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

[3] 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.

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

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

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

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

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

[25] 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.

[26] 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.

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

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

[29] 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.

[30] 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.