An Empirical Study of Build Failures in the Docker Context

Yiwen Wu*
National University of Defense Technology, China
wuyiwen14@nudt.edu.cn

Yang Zhang*
National University of Defense Technology, China
yangzhang15@nudt.edu.cn

Tao Wang
National University of Defense Technology, China
taowang2005@nudt.edu.cn

Huaimin Wang
National University of Defense Technology, China
hmwang@nudt.edu.cn

ABSTRACT

Docker containers have become the de-facto industry standard. Docker builds often break, and a large amount of efforts are put into troubleshooting broken builds. Prior studies have evaluated the rate at which builds in large organizations fail. However, little is known about the frequency and fix effort of failures that occur in Docker builds of open-source projects. This paper provides a first attempt to present a preliminary study on 857,086 Docker builds from 3,828 open-source projects hosted on GitHub. Using the Docker build data, we measure the frequency of broken builds and report their fix time. Furthermore, we explore the evolution of Docker build failures across time. Our findings help to characterize and understand Docker build failures and motivate the need for collecting more empirical evidence.

KEYWORDS

Docker, Build failure, Open-source

ACM Reference Format:
Yiwen Wu*, Yang Zhang*, Tao Wang, and Huaimin Wang. 2020. An Empirical Study of Build Failures in the Docker Context. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3379597.3387483

1 INTRODUCTION

Docker is one of the most popular containerization tools in current DevOps practice. It enables the encapsulation of software packages into containers and can run on any system [1]. Since inception in 2013, Docker containers have been downloaded 130B+ times[1]. The "Annual Container Adoption" report[2] found that 79% of companies chose Docker as their primary container technology.

With the widespread use and influence of Docker, many studies have been recently conducted to investigate its ecosystem [3], configuration file [6], workflow [12], and best configuration practices [10]. Those works have emerged a lot of great findings and brought many practical implications to developers, but were not designed to look into the details of Docker builds. Building is crucial to the software development process, which automates the process by which sources are compiled, linked, tested, packaged, and transformed into executable units [5]. In practices, builds often break (i.e., fail), and although this is not expected, broken builds can help developers to identify problems early before delivering products to end-users. Recently, the frequency and impact of build failures have been quantified in many contexts, e.g., C++ and Java builds [7], Ruby builds [2], and Continuous Integration (CI) builds [4, 9, 11]. However, to the best of our knowledge, little is known about the failure frequency and fix effort of builds in the Docker context.

To fill the gap in understanding Docker build failures (including frequency, fix effort, and their evolution), we present an empirical study of 857,086 Docker builds from 3,828 GitHub open-source projects. More specifically, we attempt to answer three Research Questions (RQs) in this paper:

- **RQ1: (Frequency) How often do Docker builds fail?** We find that the overall build failure rate in the Docker context is 17.8% and most of Docker projects (85.2%) in our dataset have at least one broken build. Frequently-built projects are associated with a low ratio of broken builds.
- **RQ2: (Fix effort) How long does it take to fix Docker build failures?** Broken Docker builds have a median fix time of 44.2 minutes in our study context. For each Docker project, more Docker build failures are related to longer fix time.
- **RQ3: (Evolution) How do failures frequency and fix effort evolve across time?** Overall, the failure rate and fix time of Docker builds fluctuate and gradually increase across time.

Paper organization. The rest of this paper is organized as follows: Section 2 describes the study setup. Section 3 presents our study results. Section 4 outlines the research agenda and Section 5 discusses the threats to validity. Finally, Section 6 concludes the paper.

2 STUDY SETUP

Figure 1 gives an overview of our study. Based on the RQs, we collect the Docker build data from thousands of selected GitHub projects, and perform quantitative studies on them.

Data sources. Our data collection involves mining two types of sources: (1) GitHub data, i.e., projects, using the Google BigQuery[3]; and (2) Docker Hub data, i.e., Docker builds, using the Docker Hub API. Docker Hub is Docker's cloud-based registry, containing

[1]https://www.docker.com/company, as of March 2020.
[2]https://portworx.com/2017-container-adoption-survey/

*Both are first authors and contributed equally to this work.

MSR '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387483

[3]https://bigquery.cloud.google.com/dataset/bigquery-publicdata:github_repos

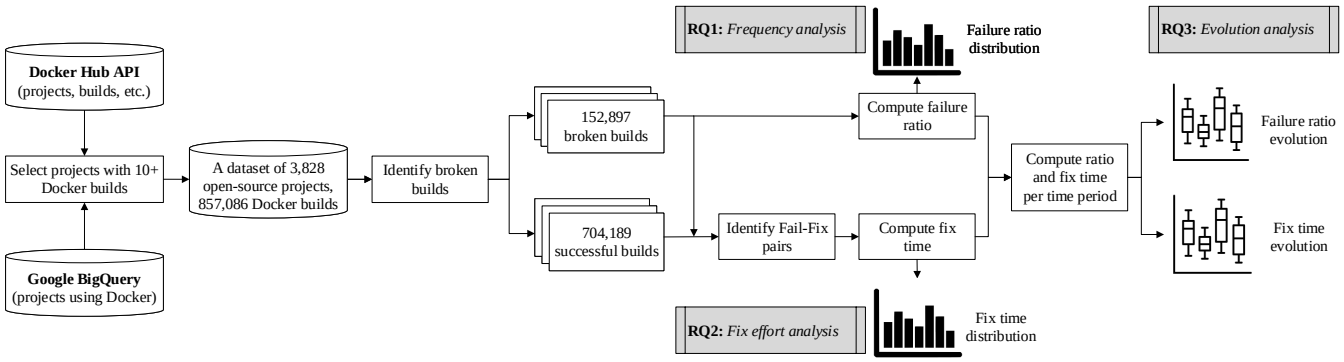

Figure 1: An overview of our study.

Table 1: Basic statistics of studied projects.

Statistic	Mean	St.Dev.	Min	Median	Max
#Total builds	224	883.9	10	36	30,615
#Successful builds	184	785.5	0	28	30,270
#Broken builds	40	245.4	0	5	10,500

3,011,048 Docker images as of January 2020. Docker Hub provides GitHub integration as well as some featured tools, e.g., automated builds[4], which allow developers to build their images automatically from GitHub sources. The build data on Docker Hub is available for mining, if the projects are public (using automated builds) [12].

Projects selection. In our study, we only consider candidate GitHub projects that use Docker. Inspired by Cito et al.'s approach [3], we use the Google BigQuery of GitHub data to select the projects that use Docker (i.e., including *Dockerfile* [6] in their Git repositories); this yields 527,543 Docker projects, as of July 2019. Next, according to the names of those projects, we filter out projects whose build data is not hosted on Docker Hub, using the Docker Hub API; our initial dataset contains 113,346 Docker projects.

Data collection and filtering. Since we can only get public Docker projects' build data, we identify public Docker projects by checking for the presence of the string "*is_automated*" through the Docker Hub API (True means the project is public); this yields 7,085 Docker projects. Based on those projects' information, we collect 870,580 build data using the Docker Hub API. Then, we filter out projects with less than 10 builds, as these might indicate experiments with the infrastructure rather than more serious Docker practice.

After this filtering, we obtain our final set of 3,828 projects for the quantitative study. In total, our dataset contains 857,086 Docker builds, including their metadata (i.e., project name, build id, build status, creation time, and end time). Table 1 presents basic descriptive statistics over the 3,828 projects, where "#" indicates the number of. On average, each project has 224 builds (median: 36). Specifically, each project has an average of 184 successful builds (median: 28) and 40 broken builds (median: 5). We note that the variance in the # of builds is very high, which may be due to differences in project characteristics, build environments or build strategies, to be further explored in future work.

3 ANALYSIS AND FINDINGS

Here, we present our study results with respect to the three RQs.

[4]https://docs.docker.com/docker-hub/builds/

3.1 RQ1: Frequency analysis

Motivation. The initial RQ aims at understanding how often Docker builds fail. Previous studies found a median of 37.4% of C++ builds and a 29.7% of Java builds at Google to be failing [7], and a 13.2% in the Visual Studio Context [5]. However, even though Docker is widely used in the open-source community [3], little research has been conducted on the Docker build failures. Thus, investigating the frequency of broken builds will help us to characterize the significance of build failures in the Docker context, and motivate the importance of our study.

Approach. To address RQ1, we first identify the broken builds by checking the build status code (less than 0 means build fails). Then, we compute the total number of broken builds and normalize it by the total number of builds, i.e., $\frac{\#brokenbuilds}{\#builds}$. Specifically, in this study, we compute the overall rate of build failures and project-specific rate of build failures.

Results. In the Docker context, 152,897 of the 857,086 (17.8%) studied Docker builds fail, which is above the 11% as reported by Zhang et al. [11] (in the CI builds context) but below the 28.5% as reported by Seo et al. [7] (in the Java builds context). As discussed in the previous study [5], build failure rates are highly sensitive to changes in context. Further, we find that 3,260 (85.2%) projects have at least one broken build; 389 (10.2%) projects have at least 50% of broken builds. Thus, build failures are very common in the Docker context, and affect most of the open-source projects.

Figure 2 shows the distribution of project-specific build failure rates in our context. Of the 3,828 projects, 568 (14.8%) projects do not have any broken builds; 2,055 (53.7%) projects have less than 20% of broken builds; and 1,205 (31.5%) projects have more than 20% of broken builds. We find that on average, the project-specific build failure rate is 19.3% (median rate: 10.5%).

Moreover, we analyze project's number of builds to investigate whether it correlates to the ratio of broken builds. We find that projects whose ratio is between 0% and 20% have a larger median number of builds (58 builds) than the projects whose ratio is 0% (16 builds) and than the projects whose ratio is between 20% and 100% (31 builds). Thus, frequently-built projects are associated with a relatively low ratio of broken builds, and the differences are statistically significant (Kruskal-Wallis test, p<0.001). This is similar to the findings reported in the CI builds context [11]. Many contextual factors may affect the build failure rate, e.g., code frequency, programming language, system and team size. With much more

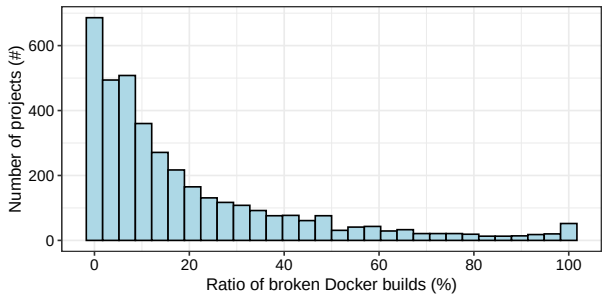

Figure 2: Distribution of broken Docker builds.

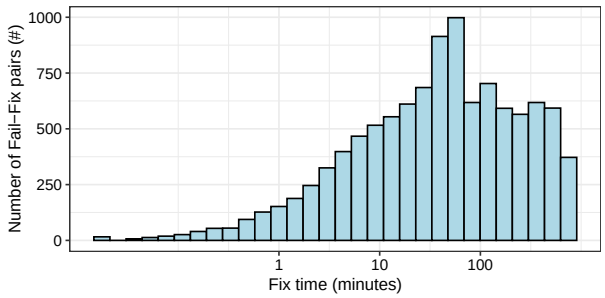

Figure 3: Distribution of fix time of broken Docker builds.

data and careful factor classification along different dimensions, some patterns may become apparent.

> 17.8% of Docker builds in our dataset fail, affecting 85.2% of projects. Also, 31.5% of projects have more than 20% of broken builds. However, frequently-built projects are associated with a relatively low ratio of broken builds.

3.2 RQ2: Fix effort analysis

Motivation. This RQ aims at understanding the time spent by developers fixing the Docker build failures. In the context of Google, developers spend a median of 5 and 12 minutes fixing C++ and Java build failures, respectively [7], and the fix time is 2.7 minutes in the Visual Studio Context [5]. Zhang et al. [11] reported that the median fix time in the CI context ranges from 18 minutes to 97 minutes. Yet we lack enough quantifiable evidence of how much effort is necessary to fix the broken Docker builds. Thus, investigating how long broken builds take to fix will deepen our understanding of the impact of broken builds on Docker-based software development.

Approach. To address RQ2, we follow the method proposed by Rabbani et al. [5] to identify Fail-Fix Pairs. First, we iterate over a project's build sequence to find a broken build (A) that follows immediately after a successful build. Next, we continue along the build sequence, skipping every other broken build until we encounter a successful build (B). Finally, we identify the Fail-Fix Pair $\{A, B\}$. This process is repeated until the end of the sequence is reached, and is repeated for each project's build sequence in our dataset. Following Google's study [7], we compute the fix time as the time interval between the finishing time of the broken build (t_{fail}) and the creation time of the successful build (t_{fix}) in the Fail-Fix pair, i.e., $fix\_time = t_{fix} - t_{fail}$. To fairly compare our results, as suggested in Google's study [7], we filter out fix times greater than 12 hours to avoid compounding the fix time with developers' schedules.

Results. Figure 3 gives the distribution of resolution time of broken Docker builds. Of the 10,566 valid Fail-Fix pairs, 511 (4.8%) pairs take less than 1 minute to fix; 2,105 (19.9%) pairs take 1 to 10 minutes to fix; 4,511 (42.7%) pairs take 10 to 100 minutes to fix; and 3,431 (32.5%) pairs take more than 100 minutes to fix. On average, broken builds in our dataset take an average of 124.5 minutes to fix (median time: 44.2 minutes), which is much longer than in Google's study (less than 12 minutes) [7]. Many factors may contribute to this difference, e.g., work environments and tooling. Also, this large difference may be due to Google requiring developers to respond quickly to failures to avoid affecting a wide range of developers [11], however, it is difficult for developers of open-source projects to fully adhere to such regulation. Moreover, unlike the traditional build process, each Docker build needs to complete the process from code to packaging, testing, and pushing to cloud registry, which may take a long time. This may bring a negative impact on the fix of the Docker build failures, because developers may wait a long time to see the final build result to determine whether the failure has been fixed. Thus, build prediction techniques are needed to remind developers what the potential build outcomes are, e.g., failure or success, and estimated fix time if the build fails.

Further, we analyze the relationship between number of broken builds and median fix time per project. We find that number of broken builds per project has a positive correlation with median fix time (Spearman's rho=0.28, p<0.001). Thus, projects with more broken builds are associated with longer fix time. Too many failures may distract the developer's time and focus, as a result, some difficult or important failures cannot be fixed in time.

> The median fix time of Docker broken builds is 44.2 minutes. A project that shows presence of large number of broken builds is related to longer fix time of build failures.

3.3 RQ3: Evolution analysis

Motivation. This RQ aims at understanding whether the ratio (fix time) of broken builds is a constant value or fluctuates across time. Prior works [5, 7, 11] mainly focus on the static analysis of build failures, few studies have investigated the evolution of build failures, the only exception being the study by Zolfagharinia et al. [14]. In that paper, the authors report on an empirical study of the evolution of CPAN build failures and find that the median build failure ratio decreases superlinearly across time. Whether the evolution of build failures differs in the context of Docker? With this RQ, we attempt to help developers better understand how Docker build failures spread and how fix efforts change over time.

Approach. We aggregate our dataset per period of six months to obtain the evolution of build failure ratio (fix time) across time. For each time period, we compute the build failure ratio per project and the fix time per Docker build. Note that we do not consider the Docker builds that were created in 2013, since there are only 63 builds in our dataset.

Results. Figure 4 and Figure 5 show the distribution of failure ratio and fix time in different time periods, respectively. The red regression line shows the corresponding evolutionary trend over the eleven studied time periods (from 2014a to 2019a). We find that,

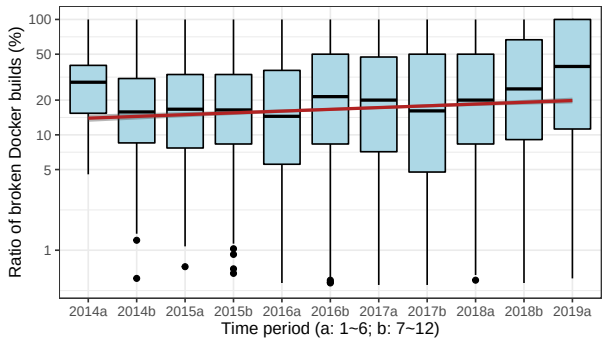

Figure 4: Distribution of failure ratio in six-month periods.

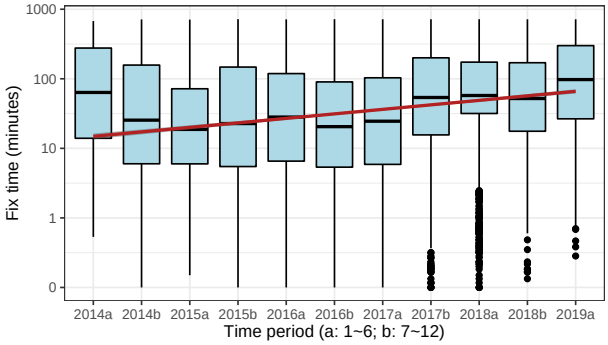

Figure 5: Distribution of fix time in six-month periods.

the median build failure ratio decreases from 28.6% in 2014a to 15.8% in 2014b. While the median ratio increases across time from 16.7% in 2015a to 36.9% in 2019a. As for the fix time, the median build fix time decreases across time from 64 minutes in 2014a to 19 minutes in 2015a. While the median fix time increases across time from 22 minutes in 2015b to 97 minutes in 2019a. However, as the regression lines show, the overall build failure ratio and fix time have slightly increasing trends, especially when considering the logarithmic y-scale. These findings are very different from Zolfagharinia et al. [14], thus verifying that build failure rates and fix time are highly dependent on the study context.

The trends that we observe in the Docker context may be due to several reasons. E.g., previous study [12] verified that Docker build latency tends to increase over time. A time-consuming and burdensome build process may increase the probability of build failure. Besides, as we discussed in RQ2, more broken builds are related to longer fix time. These factors may make the build failure rates and fix time gradually increase over time.

> Overall, the failure rate and fix time of Docker builds fluctuate and gradually increase across time.

4 RESEARCH AGENDA

As a preliminary study, our work motivates the need for collecting more empirical evidence of Docker build failures. Some initial ideas for further study include:

- Mining additional contextual data to perform a more detailed root cause analysis of Docker build failures, e.g., summarizing the common types of failures from Docker build logs;
- Exploring the relationship between Docker build failures and potential influencing factors, e.g., OSes/environments, developer experience, task complexity, build strategy, system and team size;
- Comparing the occurrences of Docker build failures between open-source projects and industrial organization projects, e.g., failure frequency and failure types;
- Creating sophisticated management tools that enhance Docker build process and integrate with different IDEs (via plug-ins), e.g., tools to predict build failures and potential fix time.

5 THREATS TO VALIDITY

Internal validity. In our study, the computed fix time for each broken build may not precisely reflect the time developers spend in fixing failures as developers often participate in multiple development tasks and switch between tasks [8]. To eliminate this threat, we follow previous works [7, 11] and exclude Fail-Fix pairs whose fix time is greater than 12 hours. Also, we note that throwing away these longer fix time may lead to some interesting data being lost. We plan to expand the experiment with more thresholds (e.g., 24 hours, 3 days, and 7 days) in the future work. Moreover, while we study the frequency and fix effort of build failures in the Docker context, we did not perform in-depth analyses of the contextual factors (and other potential confounding factors) that may affect the failure/success of a given Docker build, e.g., code frequency, system and team size, task complexity, project's programming language. In our future work, we plan to manually analyze the error messages in the Docker build logs.

External validity. Although our dataset consists of 3,828 projects with a total of 857,086 Docker builds, the results may not represent all real-world Docker builds. In particular, we can only collect the build data from the public Docker projects on Docker Hub. Also, we only consider Docker projects that are on GitHub. Thus, our findings cannot be assured to generalize to projects hosted on other services, e.g., Bitbucket and GitLab, although there is no inherent reason why they would be biased. Moreover, we only analyze open-source software. Docker build failures may be different in closed source environments.

6 CONCLUSION

Executing the build process (i.e., building) is crucial to the software development process. Many prior studies have been published describing build failures in different contexts, to the best of our knowledge, this study is the first to conduct a large-scale empirical study of build failures in the Docker context. We quantify and analyze Docker build failures, including frequency, fix effort, and their evolution based on 857,086 Docker builds from 3,828 GitHub open-source projects. Our findings motivate the need for collecting more empirical evidence to better understand how developers build Docker containers and to guide future process refinements and tool development to improve Docker building efficiency.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments on this paper. This work was supported by Program of A New Generation of Artificial Intelligence 2030 (Grant No.2018AAA0102304) and Foundation of PDL (Grant No.6142110190204).

REFERENCES

[1] Charles Anderson. 2015. Docker [software engineering]. *IEEE Software* 32, 3 (2015), 102–c3.

[2] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 356–367.

[3] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C Gall. 2017. An empirical analysis of the Docker container ecosystem on GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 323–333.

[4] Md Rakibul Islam and Minhaz F Zibran. 2017. Insights into continuous integration build failures. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 467–470.

[5] Noam Rabbani, Michael S Harvey, Sadnan Saquif, Keheliya Gallaba, and Shane McIntosh. 2018. Revisiting" Programmers' Build Errors" in the Visual Studio Context. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 98–101.

[6] Gerald Schermann, Sali Zumberi, and Jürgen Cito. 2018. Structured information on state and evolution of dockerfiles on github. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*. ACM, 26–29.

[7] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, 724–734.

[8] Bogdan Vasilescu, Kelly Blincoe, Qi Xuan, Casey Casalnuovo, Daniela Damian, Premkumar Devanbu, and Vladimir Filkov. 2016. The sky is not the limit: multitasking across GitHub projects. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 994–1005.

[9] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. 2017. 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)*. IEEE, 183–193.

[10] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. 2020. Characterizing the Occurrence of Dockerfile Smells in Open-Source Software: An Empirical Study. *IEEE Access* 8 (2020), 34127–34139.

[11] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. 2019. A large-scale empirical study of compiler errors in continuous integration. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 176–187.

[12] Yang Zhang, Bogdan Vasilescu, Huaimin Wang, and Vladimir Filkov. 2018. One size does not fit all: an empirical study of containerized continuous deployment workflows. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 295–306.

[13] Yang Zhang, Huaimin Wang, and Vladimir Filkov. 2019. A clustering-based approach for mining dockerfile evolutionary trajectories. *Science China Information Sciences* 62, 1 (2019), 19101.

[14] Mahdis Zolfagharinia, Bram Adams, and Yann-Gaël Guéhénuc. 2017. Do Not Trust Build Results at Face Value-An Empirical Study of 30 Million CPAN Builds. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 312–322.