Continuous Integration Theater in GitHub Actions

Joel Rorseth
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
joel.rorseth@uwaterloo.ca

Abstract—**Having been adopted as a standard development practice in many open-source software projects, continuous integration (CI) provides many benefits when its practices are employed effectively. However, these well-established benefits are easily negated when the principles of CI are not adhered to. In this study, we empirically analyze the prevalence of this neglect, dubbed *Continuous Integration Theater*, across open-source GitHub software projects that employ the GitHub Actions CI tool. Specifically, we analyze 1,156 projects to quantify four CI theater anti-patterns, namely infrequent commits to mainline, poor test coverage, lengthy broken build periods, and lengthy builds. We determine that commits are infrequent in 78.03% of studied projects, and that the average test coverage is only 68.37%. However, the duration of builds and broken build periods are not typically excessive, nor are they particularly common. Our analyses do reveal significant disparity between projects of different programming languages, with respect to different CI theater anti-patterns and project sizes.**

Index Terms—**continuous integration, bad practices, test coverage, GitHub Actions**

I. Introduction

As described by prominent software developer Martin Fowler, CI is a practice where members of a software development team integrate their code changes into a shared repository at least daily, each time triggering an automated build and tests to verify the integration [1]. As a result of its popularity, many tools have emerged to support CI through the automation of build compilation, test execution, and other related processes.

While CI has long been revered for its potential to reduce integration problems and hasten software releases [1], the mere adoption of a CI tool does not imply adoption of proper CI practices. Furthermore, recent work has confirmed that purported benefits of CI may be negated when proper CI practices have not also been adopted [2]. In an effort to formalize this neglect, Thoughtworks Technology Radar recently coined the term *CI Theater* [3] to describe four common CI anti-patterns, namely infrequent integration into the mainline (master branch), poor test coverage, builds remaining broken for extended periods, and using CI only for branches other than mainline. Further research of the CI theater problem is critically important, in order to understand common anti-patterns that erode the benefits of CI, which in turn waste time and money.

In this paper, we perform an empirical analysis of open-source GitHub software projects utilizing GitHub Actions for CI, in an attempt to quantify the prevalence of CI theater anti-patterns. Our study is an ideal approach to gather statistics on CI theater, as we analyze a large sample of real-world projects, and analyze metrics across several relevant dimensions. Our findings have major implications to practitioners and developers, whom stand to learn about proper CI principles for a modern CI tool, and for researchers, whom will find further evidence towards widespread abuse of CI anti-patterns. For transparency, a replication package containing all scripts utilized in this study is available on GitHub [4].

II. Related Works

Previous works have proposed and studied a wide variety of bad CI practices [5] [6], derived both CI literature and from developers' opinions. By surveying the literature, Duvall et al. assembled a catalog of 50 patterns and corresponding anti-patterns for various topics and phases within CI [7]. While CI theater anti-patterns, such as low commit frequency and excessive build duration, do appear in these works, many opinionated practices are also discussed, such as the management of build artifacts. In this paper, we focus on indisputable anti-patterns derived from the definition of CI, and employ empirical analysis to quantify their existence. Recent works have attempted to empirically survey bad practices of developers using CI. Zampetti et al. identify 73 'bad smells' relating to CI pipeline management and process, and gather empirical data through semi-structured interviews and mining of Stack Overflow questions [8]. Our study directly analyzes a large sample of software projects, rather than deriving information or new anti-patterns from interviews or indirect data sources.

To the best of our knowledge, only one other study has empirically evaluated CI theater anti-patterns using open-source project and build data. Felidré et al. perform an empirical study of open-source GitHub projects that use TravisCI for CI [9], and empirically measure the prevalence of four CI anti-patterns derived from CI theater. In our study, we analyze open-source GitHub software projects that use GitHub Actions for CI, adopting the same CI theater anti-patterns, research questions, and analysis methodology. The study of GitHub Actions theater is particularly novel among other CI tools, due to the inherent centralization of CI and repository for GitHub users, the generous free tier pricing, and plethora of capabilities beyond traditional CI processes. Our study approaches several analyses from alternative perspectives, both to accommodate differences inherent to GitHub Actions (eg.

potential non-CI usage), and to improve upon rigid assumptions of previous work (eg. including projects that use a wide variety of programming languages).

III. Study Design

In this section, we introduce the steps taken for data collection, data preprocessing, and analysis.

A. Research Questions

In this study, we focus on four research questions, borrowed from the recent empirical study of CI theater in TravisCI projects [9]. Each research question seeks to quantify the prevalence of a particular CI theater anti-pattern. Three of the four research questions map directly to three anti-patterns from the definition of CI theater, while one additional question investigates the frequency of prohibitively lengthy CI builds.

*1) **RQ1: How common is running CI in mainline but with infrequent commits?**:* In order to facilitate *continuous* integration, members of a software team must *continuously* merge changes into the mainline. This requirement is not only inherent to the definition of CI, but is a key factor in the efficiency of merge conflict resolution. Developers may neglect to integrate their changes, or instead author changes on a separate branch, leading to difficult merges that ultimately stunt integration continuity. Understanding the prevalence of this issue is important to leaders of development teams, and developers themselves, whom may be unknowingly hindering the velocity and efficiency of their teams by tolerating infrequent mainline merges.

*2) **RQ2: How common is running a build in a software project with poor test coverage?**:* As part of the continuous integration of code changes, automated tools build code changes and execute tests, to ensure that new bugs are not introduced and merged into mainline unknowingly. Without these automated processes, CI practices would be trivial and ineffectual. Furthermore, having relatively few tests, or having tests that do not sufficiently cover (test) the code, would be equally ineffectual. Understanding test coverage across software projects is critical in determining whether CI yields any benefit to users of GitHub Actions, since few other benefits can be reaped from the continuous automation. These findings are valuable to any developer employing CI, due to the particular ease in which poor test coverage may develop and become overlooked (ie. bugs may be present, yet go unnoticed).

*3) **RQ3: How common is allowing the build to stay broken for long periods?**:* When subject to long periods where mainline builds remain broken, the velocity and release cadence of a software project may grind to a halt. Broken builds prevent frequent integration into mainline, a central tenet of CI, and by their nature, prevent releases until fixed. On the other hand, excessive broken build duration may indicate failure to incorporate feedback provided by CI, which should be utilized to prevent and mitigate issues. Although this anti-pattern is derived from the definition of CI theater, measuring its prevalence is vital to understanding whether teams incorporate CI feedback effectively. Stakeholders responsible for financing the adoption of CI, such as software managers or executives, may find these statistics particularly valuable, as they ultimately reflect teams' ability to utilize and leverage CI and CI tools.

*4) **RQ4: How common are long running builds?**:* Among other potential benefits, CI aims to improve team velocity, and provide rapid feedback for code changes via automated build and test execution. By definition, slow builds are incapable of providing rapid feedback, and intuitively impair the frequency of mainline integration. As established in the literature, slow CI feedback may force developers to switch to other tasks, consequently degrading team velocity and efficiency [10]. Several works concur that quick builds should take no longer than 10 minutes [1] [11], which we will employ as a threshold in our analysis. Measuring the prevalence of long running builds is especially important in understanding whether speed-related benefits of CI are observed in practice. Therefore, stakeholders responsible for the expensive adoption and maintenance of CI stand to benefit from this analysis, as this analysis illustrates both the costs (ie. resource utilization) and rewards (ie. time until feedback) of CI in software projects.

B. Data Curation

To obtain an initial seed of projects for our study, we obtained a snapshot of GHTorrent [12], a curated offline mirror of GitHub data mined from the GitHub API. The snapshot is dated *March 6, 2021*, which falls well outside the 2019 launch window of GitHub Action. This 539GB archive contains detailed metadata for over 189 million GitHub repositories.

In an effort to augment the static project data from GHTorrent, we query GitHub API to retrieve GitHub Actions workflow files (used for CI configuration) and run history (equivalent to build logs). The Coveralls API is also queried to obtain test coverage information (for RQ2), though only for projects actively utilizing the Coveralls service. The initial set of studied projects, derived from the GHTorrent seed dataset, must be filtered to remove those unsuitable for our study. We follow a multi-stage filtering approach to remove irrelevant GitHub projects, using a combination of GHTorrent data and GitHub API data. A final set of 1,156 projects (with workflow runs) is obtained for our study. Fig. 1 illustrates the stages of preprocessing employed, which are described in the following subsections.

*1) **Stage 1: Remove Projects With No Member Data**:* Although GHTorrent has records of 189,524,128 projects (in the *Projects* table), only 9,549,783 unique projects have associated project membership records (in the *Project Members* table). These records denote the number of members (active contributors) for a given project repository, and are necessary to filter projects by team size in the Stage 2. Thus, we reduce our initial set of projects to those with membership records.

*2) **Stage 2: Remove Projects Having Only A Single Member**:* Since the lack of integration complexity in single-developer software projects precludes most benefits of CI, we choose to remove single-member projects.

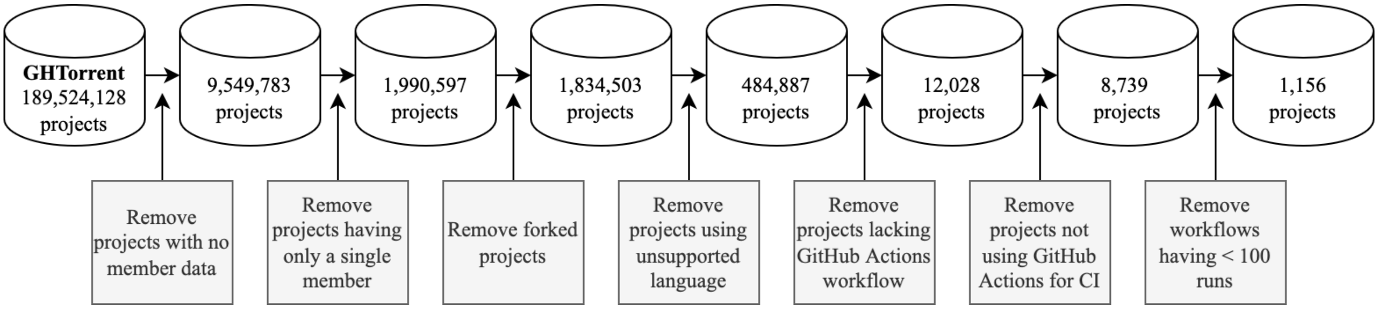
GHTorrent
189,524,128 projects
9,549,783 projects
1,990,597 projects
1,834,503 projects
484,887 projects
12,028 projects
8,739 projects
1,156 projects
Remove projects with no member data
Remove projects having only a single member
Remove forked projects
Remove projects using unsupported language
Remove projects lacking GitHub Actions workflow
Remove projects not using GitHub Actions for CI
Remove workflows having < 100 runs

Fig. 1. The filtering process, which remove unsuitable projects from our study.

*3) **Stage 3: Remove Forked Projects**:* To eliminate the possibility of statistical bias in our analyses, we remove projects that are forks of other projects. This data is also recorded by GHTorrent, and is performed offline.

*4) **Stage 4: Remove Projects Using Unsupported Programming Languages**:* For our study, we chose to focus only on Ruby, Java, C / C++, JavaScript / TypeScript, and Python projects. We use the programming language labels given to each project in the GHTorrent dataset.

*5) **Stage 5: Remove Projects Lacking GitHub Actions Workflow File**:* In this stage, we begin to collect GitHub Actions build data. In the Actions tool, configurations are defined in *workflow files*, and specify various sequences of actions and their triggers. Workflow files must be valid YAML files (ie. *.yml* or *.yaml* extension), and must be placed within the *.github/workflows/* directory in the project. For each project, we query the GitHub API to check for the existence of workflow files on the mainline branch. We remove all projects that do not have any workflow files.

*6) **Stage 6: Remove Projects Not Using GitHub Actions for CI**:* Within workflow files, actions may be specified as arbitrary Bash commands, selected from pre-built actions available on GitHub Marketplace, or imported from other scripts in the project. Due to this flexibility, Actions is often employed for purposes beyond CI, such as issue management, pull request management, or communication with third-party services. To ensure that all studied workflows facilitate CI, we filter out workflows that do not explicitly execute an established build or test command. We query the GitHub API to obtain the YAML content of all workflow files remaining after the previous stage. YAML files are subsequently parsed to extract Bash commands, which are analyzed using predefined regular expressions that identify popular usage of build and test tools, for supported programming languages (eg. Gradle, NPM, CMake).

*7) **Stage 7: Remove Workflows Having Less Than 100 Runs**:* In the final stage, we query the GitHub API to obtain the 100 most recent runs (ie. build logs) for each remaining workflow. We then remove workflows with fewer than 100 total runs, guaranteeing a large sample of build statistics for workflows and projects during the analysis. As in the previous stage, projects having no remaining workflows (after filtering) are removed from the study.

IV. Study Results

In this section, we discuss our analysis process for each research question, and present evidence to support our observations from the study. Where appropriate, we discuss trends with respect to programming language and project size. Project sizes are defined relative to the number of project members, with 2 members for *very small*, 3-4 for *small*, 5-9 for *medium*, 10-19 for *large*, and 20+ for *very large*.

A. RQ1: How common is running CI in mainline but with infrequent commits?

To answer this research question, we analyze mainline commits across all projects, and determine how many projects commit frequently (as opposed to infrequently). We first build a commit timeline for each project, by extracting the head commit timestamps from all workflow runs. Since we retrieve a fixed number of the most recent workflow runs in the preprocessing phase, we ignore those occurring on the potentially-partial oldest and newest dates. We calculate the average daily commit rate for each project, then calculate the average daily commit rate across all projects. We utilize the latter as a threshold, such that projects whose average daily commit rate is less than the threshold are deemed *infrequent*. Finally, we determine and analyze the proportion of frequent vs. infrequent committing projects.

Our analysis yields an average daily commit rate of 1.18 across all projects. A significant factor impacting this rate is the number of days with 0 commits, which if ignored during the calculation, would result in a significantly higher average of 3.15 commits per day. This suggests that many teams have many days without a single commit to mainline, and therefore violate the *daily committing* tenet of CI. Further evidence to this neglect is apparent in the proportion of infrequent projects. We find that 78.03% of projects commit infrequently on average, leaving only 21.97% whom commit more than 1.18 times per day. These low rates are consistent across programming languages and project size, though projects do exhibit slightly higher commit averages as project size increases.

In summary, an average of 1.18 commits are made to mainline per day, across all studied projects, demonstrating conformance to the daily committing expectation for CI (though not by any impressive amount). 78.03% of projects fail to commit more than 1.18 times per day on average, however,

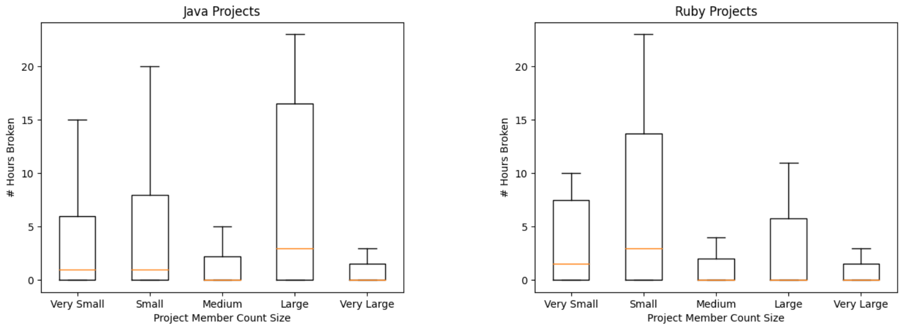

Fig. 2. Duration of broken build periods, shown separately for Java projects and Ruby projects.

suggesting that only a few very-frequent committers (outliers) truly commit more than once daily.

B. RQ2: How common is running a build in a software project with poor test coverage?

In this analysis, we query the Coveralls API to retrieve coverage reports. Each report expresses the total test coverage as a percentage, and is specific to the project code snapshot in a certain commit. To ensure that coverage statistics are representative of the workflow runs collected, we seek reports for mainline commits occurring no earlier than 7 days before the creation date of the most recent workflow run. From these reports, we select the most recent. In total, only 23 of the 1,156 selected projects have Coveralls records meeting these criteria. This set consists of 4 Java projects, 10 JavaScript / TypeScript projects, 6 Python projects, 2 Ruby projects, and 1 C / C++ project. Finally, we calculate average test coverage over all projects, as well as when grouped by programming language.

Across all projects, we find that the average test coverage is 68.37%, with a median of 77.93%. This suggests that several outliers exist with particularly low coverage values (ie. the distribution is skewed to the left), which means that several projects have very low test coverage. Over all projects, the standard deviation is 29.41%, which further attests to the noteworthy variance among coverage values. Overall, these statistics indicate that many projects do retain adequate coverage, though many outliers significantly neglect test coverage, and therefore proper CI principles.

These findings are reinforced when grouped by programming language, which illustrates significant variance in this dimension as well. Having the best reported coverage, JavaScript / TypeScript projects have an average of 84.47%, median 86.94%, and standard deviation of 12.84%. Moreover, the minimum coverage was 60.95%, while the maximum was 97.88%. On the other hand, Python projects have an average of 49.79% and median of 52.93%, with standard deviation of 29.81%. Despite similar sample sizes, Python variance is more than double that of JavaScript / TypeScript, and has several outliers (including at least one value of 0%). Across all languages, only JavaScript / TypeScript and Ruby projects do no practice CI theater with via coverage neglect, while Java and Python projects exhibit poor averages.

In summary, much variance exists across programming languages, with Java and Python projects having less than 50% average coverage. For other languages, and across all languages, average coverage is adequate though not good. Thus, CI theater not uncommon with respect to the insufficient test coverage anti-pattern.

C. RQ3: How common is allowing the build to stay broken for long periods?

To answer this research question, we first extract the head commit timestamp and conclusion (for our purposes, success or non-success) associated to each workflow run, in each project. We then create a timeline of timedeltas (elapsed time between two timestamps) for the workflow, to capture the elapsed time during all observed periods of unsuccessful runs. We combine timedeltas across all workflows and projects, then calculate statistics across project timedeltas, and then again across programming languages and project size.

First, we calculate a duration threshold, equal to the 3rd quartile timedelta across all projects. Workflow run (build) timedeltas exceeding this threshold are deemed *excessively long*. We calculated a threshold of 22:41:54 (denoted as *hours:minutes:seconds*), and found that 51.99% of projects have at least one run exceeding it. However, the average broken duration across all projects (8345 timedeltas) is 82:27:37 (approximately 3 days 10 hours), while the median is only 1:22:32 (approximately 1.5 hours). The average is significantly skewed by high outliers, with a standard deviation exceeding 16 days, and maximum duration exceeding 491 days. Therefore, we assert that excessively long broken builds are not common in studied projects.

When grouped by programming language, as in Fig. 2, projects medians are very similar and close to the overall median. The average broken build duration is higher for Java and Ruby projects than across all projects, by approximately 1 day, and is lower for Python projects by approximately 12 hours. This suggests that Java and Ruby projects are more neglectful for fixing broken builds, and for demonstrating CI theater. The standard deviation across Java and Ruby

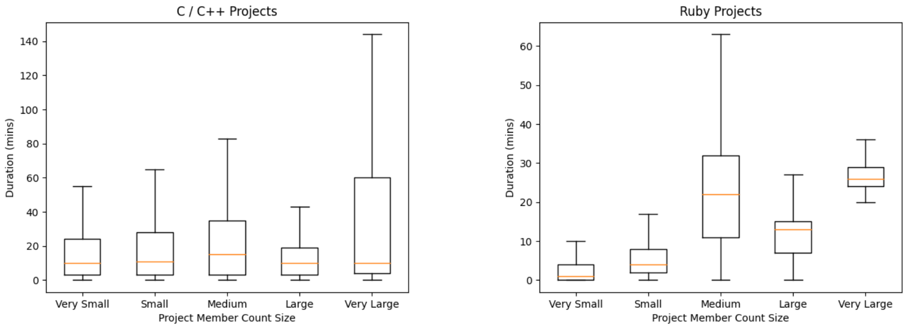

Fig. 3. Duration of builds, shown separately for C / C++ projects and Ruby projects.

timedeltas exceeds those of all other languages as well, with values of 465:09:16 (approximately 19 days 9 hours) and 534:59:16 (22 days 7 hours) respectively.

In summary, we observe that only 51.99% of studied projects have one or more lengthy broken builds, yet most broken builds are fixed quickly. While this CI theater criteria is not common overall, Java and Ruby projects are significantly more neglectful of broken builds than other languages.

D. RQ4: How common are long running builds?

To answer our final research question, we utilize a 10 minute build duration threshold established in the CI literature [1] [11], where workflow runs (builds) whose length exceeds this threshold are deemed (excessively) *long running*. In our analysis, we extract and aggregate the duration of each workflow run across all workflows, for each project. We then calculate the proportion of projects having at least one run exceeding the 10 minute threshold. In addition, we produce statistics and plots describing summarizing duration all projects, as well as when grouped by programming language and project size.

Across all projects, 81.23% have one or more runs exceeding the 10 minute threshold. Similar to the criteria in other research questions, much variance is exhibited across all projects, with a standard deviation of 18:37:49. While the average build takes 53 minutes 23 second, the median build takes only 6 minutes 20 seconds. Thus, while the majority of projects have tolerated long running builds in the past, the majority of builds do finish quickly.

Most notably, JavaScript / TypeScript projects contribute disproportionately to this variance, having the highest average duration across all languages with 1 hour 16 minutes, as well as the highest standard deviation of 24:57:39. By contrast, the lowest average duration of 27 minutes 36 seconds belongs to Java projects. Several projects from all languages have outliers that greatly exceed 10 minutes, and therefore fall victim to CI theater. Of all language groups, only Ruby and C / C++ median durations exceed the threshold, with 16 minutes 11 seconds and 11 minutes 17 seconds respectively. Since more runs in these languages exceed the threshold than not, we assert that projects using Ruby or C / C++ tend to neglect this aspect of CI theater. Fig. 3 illustrates that duration generally increases as project size increases, which is particularly pronounced among Ruby projects.

In summary, 81.23% of projects have experienced at least one long running build, though most builds finish well below the 10 minute threshold. Builds in Ruby and C / C++ projects typically exceed the threshold, indicating that projects using these languages typically practice CI theater instead of CI.

V. Discussion

In our analyses, we present empirical evidence quantifying the neglect of CI principles in real-world GitHub Actions projects. All four CI theater criteria (from the research questions) are present in many studied projects, indicating that many development teams neglect one or more principles of CI. Practitioners, namely developers and managers, should pay attention to these anti-patterns, to ensure that their own CI strategy adheres to proper CI practices. Avoiding CI theater anti-patterns is essential to retaining the benefits expecting when adopting CI or a CI tool.

When viewed from a different perspective, certain anti-patterns are significantly more common to projects of particular programming languages. Practitioners should understand this evidence, which empirically suggests that certain programming languages and their frameworks present more difficulty for adhering to CI principles. Of particular interest are the metrics affected by framework-supported tools, such as test coverage (RQ2) and build duration (RQ4).

Relating to the broader theory of CI neglect, our findings echo many of those established in the literature. More importantly, when considering the similar findings of studies for other CI tools, our work reaffirms that CI tools have yet to significantly mitigate many of the CI theater anti-patterns. When compared to the recent TravisCI CI theater empirical study [9], we find that builds have become even more lengthy, code coverage has dropped, and commits to mainline have become less frequent. As the results of the aforementioned paper were largely confirming widespread neglect, our analyses highlight and reaffirm the larger trend towards neglect of CI principles. However, we find that builds are fixed significantly faster in our GitHub Actions-based projects, which suggests that some aspect of the tool, or perhaps its users, are contributing towards improved CI practices in modern software projects.

REFERENCES

[1] Fowler, M. & Foemmel, M. Continuous integration. (2006)

[2] Luz, W., Pinto, G. & Bonifácio, R. Building a collaborative culture: a grounded theory of well succeeded devops adoption in practice. *Proceedings Of The 12th ACM/IEEE International Symposium On Empirical Software Engineering And Measurement*. pp. 1-10 (2018)

[3] Thoughtworks Technology Radar, CI theatre. https://www.thoughtworks.com/en-ca/radar/techniques/ci-theatre. Accessed 2022-04-15.

[4] Rorseth, J. CI-Theater-GH-Actions. *GitHub Repository*. (2022), https://github.com/joelrorseth/CI-Theater-GH-Actions

[5] Duvall, P., Matyas, S. & Glover, A. Continuous integration: improving software quality and reducing risk. (Pearson Education,2007)

[6] Humble, J. & Farley, D. Continuous delivery: reliable software releases through build, test, and deployment automation. (Pearson Education,2010)

[7] Duvall, P. Continuous delivery patterns and antipatterns in the software lifecycle. *DZone Refcard*. **145** (2011)

[8] Zampetti, F., Vassallo, C., Panichella, S., Canfora, G., Gall, H. & Di Penta, M. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering*. **25**, 1095-1135 (2020)

[9] Felidré, W., Furtado, L., Costa, D., Cartaxo, B. & Pinto, G. Continuous integration theater. *2019 ACM/IEEE International Symposium On Empirical Software Engineering And Measurement (ESEM)*. pp. 1-10 (2019)

[10] Gallaba, K., Junqueira, Y., Ewart, J. & Mcintosh, S. Accelerating Continuous Integration by Caching Environments and Inferring Dependencies. *IEEE Transactions On Software Engineering*. (2020)

[11] Hilton, M., Nelson, N., Tunnell, T., Marinov, D. & Dig, D. Trade-offs in continuous integration: assurance, security, and flexibility. *Proceedings Of The 2017 11th Joint Meeting On Foundations Of Software Engineering*. pp. 197-207 (2017)

[12] Gousios, G. & Spinellis, D. GHTorrent: Github's data from a firehose. *2012 9th IEEE Working Conference On Mining Software Repositories (MSR)*. pp. 12-21 (2012)