CI told you

Exploring the role of testing strategies as part of CI pipelines and their impact on DevOp metrics in Open Source projects.

Kiril Panayotov

Supervisor: Shujun Huang
Responsible Professor: Sebastian Proksch
Examiner: Marco Zuniga

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfillment of the Requirements
For the Bachelor of Computer Science and Engineering

June 22, 2025

Abstract

Continuous Integration is a used extensively in modern software engineering for both proprietary and open-source projects. Many studies have studied its benefits and drawbacks, finding how it increases development productivity and stability. However, CI is a set of practices from static linting to calculating the code coverage of the underlying test suites. In order to choose whether to make use of that technology or to evaluate the overall performance of a project's development, practitioners make use of certain measurements, DevOps metrics being one of the most significant ones. We aim to analyse the effects of testing strategies within the CI over a set of DevOps metrics. This is done by collecting over 5778608 executions of GitHub CI workflows that involve a test-running step from 476 open-source projects. We see that 69.48% of runs happen after a pull request or a push. In the end, we found that frequent CI test execution didn't increase the project's DevOps metrics, indicating that developers should try limiting the unnecessary execution of tests to save on resources. Further we see lack of Pearson statistical significance for the correlation between coverage in CI and the metrics in the smaller set of selected projects.

1 Introduction

Continuous Integration (CI), a technique firstly explored by Beck Kent in his work "Extreme Programming" in 2000, [2], has seen a dramatic increase in usage given its promising benefits to the development process and enabling practitioners to deploy at a much faster rate. [17]. In 2016, Hilton M, Tunnell T, Huang K, et al gather over 34,544 open-source repositories on GitHub based on popularity (at that time) and conclude that 70% of them use CI [9]. The advantages of CI include but are not limited to improving the stability, productivity and speed of the development process [9], [17]. Understandably, the field itself has seen a lot of exploration in terms of many academic papers, systematic literature reviews and books [4], [17].

However, implementing a CI within a team environment comes with technical and cognitive cost [19]. Thus, teams have to decide if it is worth including such a tool within their development process. Decisions like this that affect projects' performance motivate the creation and adoption of metrics that measure productivity. A particular example are the DevOps metrics which practitioners have defined and used extensively [7]. In the documentation for their CI/CD tool TeamCity, JetBrains list several, which we have particular interest in: Delivery Frequency, Delivery size, Lead time to changes, Mean Time to Recovery and Defect Count [12].

Some elements of the CI usage remain unclear. In their 2022 systematic literature review , Eliezio Soares et al, suggest further research on how test effort and test quality in project influence each other. [17]. Additionally, another study gives the idea of looking at CI as a collection of practices and analyse their own effects on the project's performance [5]. Automated testing, while non-mandatory, is a commonly used step in CI [3] and it is often suggested in early CI literature [4], [6]. Further, to the best of our knowledge, the relationship between the DevOps metrics to testing within the CI in particular have not been explored. Thus, a question remains: what is the impact of testing strategies in CI on the performance of a project? With this knowledge, we can help practitioners improve projects' quality and development efficiency by arguing for and against inclusion and execution of tests within CI and the implementation of such pipelines.

This study will explore whether there is a inherent connection between certain performance metrics and the testing strategies of a project. These involve the coverage rate and the testing frequency of projects. We focus our study entirely on GitHub actions as a CI provider, over alternatives like TravisCI and Jenkins, since there has been less research focus on it. We collect data over the span of a year in the selected open-source projects in a time-series manner. In particular we want to answer:

RQ1 - What triggers CI workflows that execute tests in projects?

RQ2 - How do changes in testing frequency within CI relate to shifts in DevOp metrics?

RQ3 - What is the effect of code coverage within CI on the DevOps metrics?

To achieve this we did a time-series analysis of 478 selected open source projects by examining their DevOps metrics along with their CI workflows. We analyse their executions and compare their frequency against the DevOps metrics measurements. Finally, we analyse logs from 15 projects for detailed analysis of code coverage over time. We find that pushes and pull requests trigger 65.71% and 59.94% of CI workflows that run tests respectively. However, a substantial 32.73% execute on a defined schedule. Out of the 5778608 analysed workflow runs, 80.94% were triggered by either a `PUSH` or a `PULL_REQUEST`. We did not see statistical significance between frequent test execution and changes in the DevOps metrics, nor between them and high coverage in CI workflows. We suggest limiting unnecessary executions to preserve computational power and electricity and further research with more complex statistical analysis.

The paper will follow this structure: first, we give a summary of the related work in this field with relevant conclusions from similar academic publications. Then Section 3 presents the important background for the study. Section 4 focuses on our used methodology and project selection strategy. Sections 5, 6 will show the results and discussion on their implications respectively. Finally, sections 7 and 8 consists of an ethical discussion, general conclusion of this paper.

We collect the data for this project using a tool we created - *CI-tool-du*. The data and the tool are available on Zenodo.org under 10.5281/zenodo.15711349 and 10.5281/zenodo.15715218

2 Related Works

As a concept, Continuous Integration refers to the practice of executing a set of given tasks when integrating new code into software projects. Beck Kent introduced this practice first in "Extreme Programming" published in 2000 [2]. As it developed, the practice could involve building the project, running its tests and static code analysis amongst other techniques. This way, a reliable quality gate for code integration can be established with the belief that this will improve the stability of the development process as well as allow for more seamless integration of new features or bug fixes. The rise of popularity reflect those beliefs, as Hilton et.al. [9] conclude that 70% of the 35,544 top repositories on GitHub in 2016 used CI. The significance and rise its rise of popularity was even compared to the importance of version control by Humble et. al. [10]. It's relevance can also be seen in the private sector. JetBrains claim that CI together with Continuous Development (CD) lead to better code quality, reduced risk and downtime etc. [11].

Those effects indeed seem to be the case as concluded by multiple sources - CI does have a beneficial impact of software projects [4], [18], [9]. In 2022, Soares et al. [17] performed a systematic literature review on the CI practice and its effects providing an updated overview of the field. While many of the 101 analysed papers argue positively about its effects, 5 papers claim that there are technical difficulties with CI, especially to the environment within the build and what is executed in the configured pipelines. This is of interest to this study, as it shows the complexity of CI, further reinforcing the idea that it should be explored as a collection of practices rather than a single project attribute. Another interesting question this SLR poses is about the criteria used to distinguish if CI is used in a project - they find that 42.5% of their primary studies have not defined any. They suggest being "more restrictive regarding those (TravisCI) logs, since not every build log from TravisCI may come from a project that properly employs CI". In the Methodology section of this paper, further details are given on how we are defining if a project uses CI, particularly whether either of its configured pipelines runs tests.

Sizilio et al. [16] empirically studied the relationship between adoption of CI and testing practices. Specifically, they analysed the evolution of test ratio and coverage, as well as if there is a relationship between the adoption of CI and those trends. By comparing 82 projects with CI and 82 without, they conclude that there is a trend of increasing test ratio in 40.2% of CI projects, while only this ratio for no CI projects was only 17%. In this study however, emphasis is instead put more on the somewhat inverse direction: does high coverage in CI result in better project performance. Beller et al. [3] studied effects very close to this one. Their study explored the size and execution length of test suites in CI builds, as well as their influence on the build results. After analysing 13,590 projects using Travis CI, they concluded that 81% run tests in their continuous integration pipeline, on average 1433 tests are executed per build and test runs take around 1 minute median execution time for Java projects and 10 seconds for Ruby ones. In contrast to their research, this paper focuses on the frequency of the CI tests execution and particularly on CI pipelines ran in GitHub Actions as opposed to Travis CI. Further, We explore what events trigger those builds.

3 Background

In this section, we will give information on context important for this study. This includes the analysed CI provider on GitHub - GitHub Actions, as well as the DevOps metrics we measure.

3.1 GitHub Actions

GitHub is one of the most popular platforms for sharing open-source projects. Further, it has integrated features for collaboration - issues, pull-requests, discussions, as well as project management ones - labels, project trackers, etc. In 2018, GitHub launched its Continuous Integration and Deployment automation tool called Actions. It allows projects whose codebases are already hosted on GitHub to utilise its own CI/CD platform.

Actions allow for various automations within a project - from automated replies to and categorisation of issues to complex pipelines for linting, building and testing projects. Several benefits of this platform include free use for public repositories, virtual machine runners on various OSes and processor architectures. Additionally, pre-made solutions for popular frameworks or languages exist on an Action marketplace (e.g. pre-made workflow for executing JUnit tests, building and publishing NPM packages etc.).

Actions consist of a Workflow which executes a set of Jobs. Each Job has Steps that execute a shell command or pre-made GitHub Action script (e.g. check-out the repository within the environment of the Workflow so that it can access it). The entire Action - its execution order and each sub-element - is defined by a YAML file located is a special folder called `.github/workflows`. Files there will be parsed by GitHub upon hosting the repository on the platform.

GitHub provides with an API that allows fetching of information about those workflows. One important limitation is the fact that the information we aim to gather is limited to only 400 days in the past for workflow executions, duration of each job, etc and 90 days for concrete build logs.

3.2 DevOps metrics

To evaluate the effect of running tests within the CI on the performance of a project, we adapt the DevOps metrics from Google's DORA report [7] and JetBrain's article on CI/CD performance [12] to open source projects given their popularity and usage. Aiming to have a comprehensive measurement of projects' performance, we focus on several metrics: Delivery Frequency, Mean Time to Recovery and Defect Count. These metrics rely upon two crucial definitions - that of delivery and that of a defect. However, applying them within the open-source context is not trivial. We couldn't identify clearly what is considered "production" and what's not because such projects are usually provided as is and information about what code exactly do owners execute on deployment is not easily available. Thus, we have to translate the studied metrics in order to perform our research.

Defining delivery and defect In 2017, IEEE, IEC and ISO standardised a vocabulary of terms used in Software Engineering. There they define delivery as "release of a system or component to its customer or intended user" and defect as "imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced" [1].

Striving be as close to them as possible, we define delivery as a tag release and defect as an issue considered a bug. While the former is trivial given the provided GitHub API endpoint for tag releases, the latter becomes a relative challenge to determine correctly. Upon manual evaluation of over 7000 issue reports, Herzig et al. [8] conclude that "every third bug report is no bug report" i.e. 33.8% of all issues are wrongfully classified as bugs. The authors suggest manual human analysis of (significant part) of the data. Unfortunately, this is incompatible with the current resource and time constraints.

To mitigate this, we have chosen a strategy used by Vasilescu et al. [18]. First, they filtered in projects who have used labels and issues - at least 100 issues reported and at least 75% of them have labels. After performing manual evaluation of issue management in active (at the time) repositories, they have made a set of words related to bugs: defect, error, bug, issue, mistake, incorrect, fault, flaw etc. Thus, we parse the name and description of the labels and classifies an issue as bug if any of the said keywords appear. Further, to filter out scenarios where any of the listed words appears in a phrase indicating their revomal "e.g. bug fix" we do not consider labels that also include the words "fix" or "resolve".

Translating DevOp metrics to Open-source context Below are the translations of the metrics we calculate together with a high-level overview of the algorithm used.

Delivery frequency is calculated by measuring the total number of tag releases within each 30 days interval.

Mean time to recovery is measured as the average time to close defects (issues that are considered bugs as described above). The algorithm works by listing the issues that are defects and calculating their close time (difference between when they were created and closed). We then assign each issue to an interval based on when they were closed. In the end we average out the close time per interval and determine the MTTR.

Defect count is calculated by the number of open issues that are considered defects for a certain period. This is done by listing those issues and incrementing the defect count for every 30 day interval which is fully between the created and closed date for this defect.

Initially, we considered another popular metric - *Change Failure Rate* (CFR). However, in the end we decide to exclude this metric because of the difficulties related to adopting it for open source projects. CFR gives the percentage of deployments that resulted in a failure. GitHub allows for a deployments feature but upon manual inspection of several repositories we found that it is often only to host project documentation websites on GitHub's infrastructure. If we were to consider version releases as deployments, then defining when do we consider a version release to be a failure becomes difficult and impractical - the existence of a "patch version" within the widely used Semantic Versioning system suggests the common occurrence of minor bugs. If we consider every new patch version within a minor release, we may be looking at a case where every version upon the latest one has at least one bug implying that they are failed releases.

4 Methodology

In this section we discuss our methodology for collecting and calculating data for this research. First we describe how we detect if a CI executes a workflow, then we give an overview of the project selection strategy. Finally, we explain how the code coverage and test suite evolution was performed and the challenges related to it.

4.1 Detecting CI test execution in a project

Detecting presence of CI within project proves to be a non-trivial issue. As previously mentioned, in their systematic literature review Soares et. al. [17] find that 42.5% of their primary studies did not give an explicit definition of when a project uses CI. They argue that build logs generated by a CI provider are insufficient to imply that a project employs CI properly. Analogically, in the context of GitHub Actions, simply relying on presence of workflows is not enough to mark a project as one that uses CI. This is because there are examples of more socially-oriented or project management Workflows that automatically reply on newly posted issues or label them based on an algorithm.

We only strictly consider CI pipelines that run tests in this study. Simply parsing all YAML files within the `.github/workflows` won't be sufficient as they do not necessarily define a workflow, e.g. due to bad formatting or entirely different intention. Thus, we use the GitHub API to collect the workflows and their source YAML files. After the parsing the source files content, we specifically look at the the executed jobs and their steps. If a step includes the word "test" in its name or if its command runs any tests, we consider this workflow to be executing tests. The latter is done by matching the command on a regular expression, computed after manual analysis of several workflows. It is a combination of three sub-expressions - one per considered programming language. Each matches on a dependency manager command (e.g. `npm`, `pnpm`, `yarn`, `mvn`, `gradle`, `bun`, `dotnet`, `phpunit` ...) and then a keyword that suggests executing a test suite (e.g. "test", "verify"). By unionising matchers on the command and the name of a step, we believe that this can reliably identify workflows as ones that indeed execute any tests of the software.

4.2 Project selection strategy

We aim to find projects with sufficient and relevant data, so we filter out toy/example projects. To do this we queried public projects that are not archives or forks and have at least 50 stars [13]. We only considered Java, JavaScript, Python, TypeScript, C++, C# and PHP projects, being one the most popular languages on the platform. This resulted in 6715 projects. We removed all projects that did not have any defined workflows which resulted in 4097 projects. In order to work with maintained and active projects, we analysed the first set of 6715 projects in the number of commits for the last year. Similar to Ray et al. [15] we filtered out projects that fellow below the first quartile. We used the same technique to filter out project with low number of GitHub releases, since we use it to calculate our DevOps metrics. This left us with 1921 projects. Then, as mentioned before, filtered on project that have a minimum of 100 issues with at least 75% of them labelled. 670 projects of the initial set followed this criterium. Finally, we only considered projects that run tests in their CI with the method described above. We ended up with 478 projects and 1680 workflows for analysis.

4.3 Calculating the Test frequency, DevOps metrics and their correlation

Using the definitions described in subsection 3.2, we calculated the selected DevOps metrics for the 478 projects that got filtered-in. When calculating Test frequency we need all the executions of a given workflow. We use the GitHub API to collect each run together with when it started, what triggered it and how it concluded (success, failure, skipped, etc.). Then, by aggregating on the started date and time, we group executions for each month. We ignore all runs that are skipped when calculating the frequency. The number of executions in each month corresponds to the test frequency in that period.

We use the GitHub API for the metrics as well. To calculate the DevOps metrics with the translated definitions above, we need all the issues and releases in the analysed period of 12 months. After the data is stored locally, a script aggregates the data and calculates the metrics for each month as described in Section 3.2. The resulting measurements are stored per month for each project.

4.4 Analysis of code coverage

Calculating test code coverage of arbitrary open source projects is a non-trivial task, as also discussed by Sizilio et al. [16]: failing builds, tests or missing dependencies that are currently unavailable make the collection of this data difficult, especially given the resource and time limitations. Initially we tried randomly choosing 10 Maven Java projects from the sampled set of 478, given the well-formatted and consistent output of the Maven CLI. For each of those projects we would collect one commit per month that represents the state of the project at this point in time, with the intend of calculating the coverage at that point. However, unless we match the same test running command that is executed in the CI at this point in time, this method would have collected the coverage for all tests present in the repository.

Since we are interested strictly in the code coverage resulting from the tests ran in the CI, we opted in for collecting the coverage directly from the logs of ran CI workflows that report coverage. This method further allows us to collect data for various projects regardless of their programming language, used frameworks or tools. Unfortunately, GitHub limits the availability for those logs to 90 days in the past. Thus, we analysed shorter periods of one week resulting in 12 data points per project. We sampled a successful execution of the workflows for each week. If none was present, the last valid such was selected. We excluded projects that miss runs for more than 4 of the weeks or that do not print coverage data in their logs.

To find workflows that report on coverage we looked for workflows from our set that either have the terms "coverage" or "codecov" in their name or a name of any of their steps. For each of their runs, we parsed the downloaded the logs and collected the reported coverage by inspecting the latest of their collected runs to analyse how the coverage is reported. By adapting the template of the report in the logs, we automatically collected the data for the rest of the runs. In some cases projects had multiple workflows that detected coverage. They varied in either the dependency versions or operating system they are ran on. For this study we stick to Linux based workflows and choose tool version arbitrarily. If the coverage is reported per module/part of the project and not in one general single rate, we exclude it. In the end we collected data for 15 projects given the resource and time constraints.

5 Results

We managed to collect over 1680 workflows over the 478 filtered projects and their 5778608 runs (executions). The project `pytorch/pytorch` had abnormally more test executions then the rest so we decided to filter it out.

5.1 Execution of CI Workflows with tests

In Table 1, the results can be observed. As expected, most of the times the CI workflows with tests are executed upon a change to a defined branch - either via a pull request or a push. However, we see that more than half of the found workflows execute on schedule. That is, they execute automatically on a given time or interval. Another interesting phenomenon is that almost 60% of the workflows do not include the "workflow_dispatch" trigger, i.e. they cannot be manually executed by an operator of the repository. On the other hand, some workflows have executions that are only from manual runs.

Observation 1: Most of the CI workflows are ran on pushes (65.71%) or pull requests - (59.94%). 32.74% of them run on a schedule. Surprisingly, almost half (48.87%) of them have the option for manual execution disabled.

We can notice a striking number for the success rate of the ISSUE_COMMENT runs: only 3.55% of them are successful. This is because 96.29% of them were skipped. In the case of `actualbudget/actual`, the project contributors use this method to create a pseudo-command line where upon commenting "/update-vrt", a set of specified set of tests would execute. In `jupyterlab/jupyterlab`, the workflow would execute normally on pushes or pull requests. For comments, it will first check if they contain a command in the form of a sentence - either "Documentation snapshots updated." or "Galata snapshots updated." This will execute a set of documentation tests or a set of UI regression tests from their UI testing framework "Galata". In total there were 215266 runs that were not skipped and triggered by ISSUE_COMMENTS.

Trigger	Workflows	Total runs	Success rate
PULL_REQUEST	1104	3249632	65.50%
PUSH	1007	765093	75.47%
WORKFLOW_DISPATCH	859	40740	82.65%
SCHEDULE	550	407199	83.91%
ISSUE_COMMENT	73	984346	3.55%
MERGE_GROUP	55	72902	85.90%
RELEASE	40	1433	78.23%
WORKFLOW_RUN	23	184593	66.55%
Others	17	72670	35.06%
TOTAL:	1680	5778608	57.41%

Table 1: Workflows and concrete workflow runs by triggers. Data for 478 projects, 1680 configured workflows and 5778608 runs.

Observation 2: Project contributors use comments on issues as a trigger for test-executing CI workflows via command-like sentences.

Out of the 4794320 analysed runs that were not triggered by an ISSUE_COMMENT event, 69.20% were successful (their `Conclusion` field was ”SUCCESS”). `PULL_REQUEST` event triggered 56.24% of the runs, while `PUSH` triggered 13.24% . However, the success rate for the former was nearly 10% lower than the latter - 65.50% vs 75.47%. Scheduled runs enjoyed one of the highest success rates of 83.91%.

Observation 3: Almost 70% of all runs were triggered by pull requests. At the same time, 75% of the runs triggered by pushes were successful.

When it comes to the `WOKRFLOW_DISPATCH` option, we observe an interesting occurrence. For several workflows, we observed how the option was not included in their original YAML configuration file, and was only added later. For the ”Overall tests” of `Avaiga/taipy`, the option was only introduced after a problem where the CI wasn't ran on pull requests initiated by `dependabot` (an automatised system for detecting vulnerable package versions). In fact, many of the provided pre-configured GitHub Action templates (e.g. a default pipeline configured for NodeJS that builds and runs tests) do not have the `WORKFLOW_DISPATCH` option enabled.

Upon further inspection, we see that for the 859 workflows that enable this feature, 482 never had a single manual execution. When looking at the executions for all of them, we see that only 6% were manual. In total contrast, 34 workflows are solely ever executed manually. One such example is a CI for executing Regression tests in `airbytehq/airbyte`. In another case, the manual runs were actually triggered by a bot. For `Avaiga/taipy` the workflow that had only manual executions was an automated script to publish the project on the python package index.

5.2 Frequency of test execution VS DevOp metrics

We did not find any strong Pearson correlation between the testing frequency and the selected metrics. Figure 1 shows box-plots with the resulting correlations from all projects. We see that, on average, higher test execution frequency results in more slightly more defects, deliveries and lower mean time to recovery. However, in Figure 2 we see that this correlation is not significant most of the times. In fact, this is the case in only 7.6% of the projects when it comes to the effect on Delivery Frequency. The numbers are 19.3% and 7.8% for Defect Count and Mean Time To Recovery respectively.

Observation 4: There is no Pearson statistical significance between the frequency of test executions within CI and the Defect Count, Delivery Frequency, and Mean Time To Recovery metrics.

5.3 Code coverage and test suites VS DevOps metrics

We did not find strong Pearson correlation between the coverage and DevOps metrics amongst the 15 analysed projects. Figure 3 shows the correlations while Figure 4 - the p-values. One project `bloomberg/memray` further used the coverage rate as a quality gate and required a minimum of 90%

We make an interesting observation in `nodejs/node`: the project observes its coverage on different platforms: windows and linux. We observe differences in each coverage report with the Windows one getting 2.77% less on average between the months compared to the Linux report. Further, we investigated the source YAML files of both workflows to see that the exact same events trigger them. The only slight difference was between the files which when changed on push trigger the workflow one of which is platform dependent - `Makefile` vs `vcbuild.bat` for compiling C++ code.

Additionally, 13 out the 15 configured workflows, upload their coverage reports to either GitHub, Codecov or another external tool for analysing coverage over time. In the case of `microsoft/semantic-kernel`, they save the report as an artefact on GitHub. Upon completion of the workflow that performs this task, another one picks up the report, parses its contents and publishes it as a comment underneath the pull request that triggered the execution in the first place.

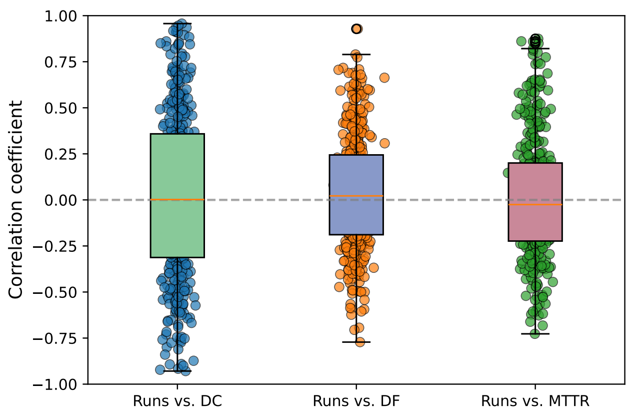

Figure 1: Box plot of correlations between testing frequency and DevOps metrics over all projects.

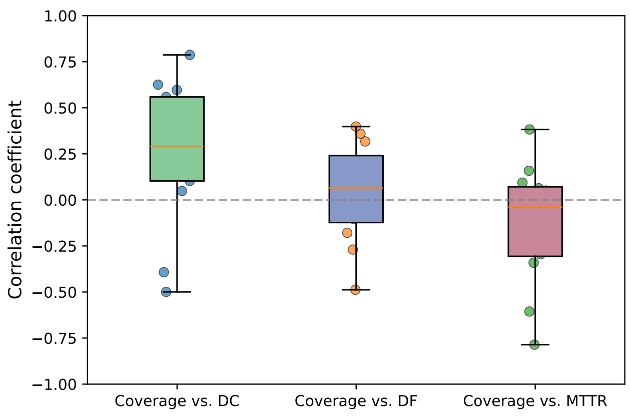

Figure 3: Box plot of correlations between CI coverage and DevOps metrics over all projects.

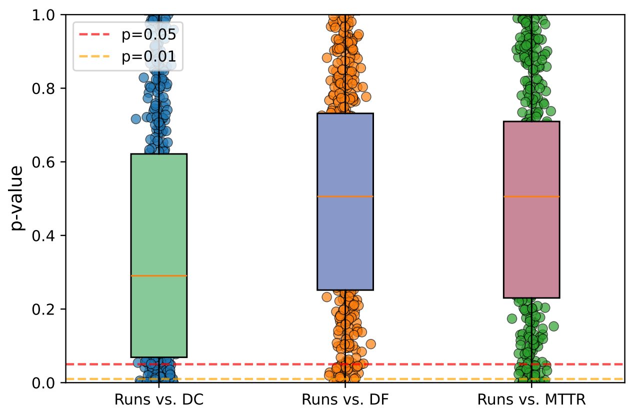

Figure 2: Box plot of p-values between testing frequency and DevOps metrics over all projects.

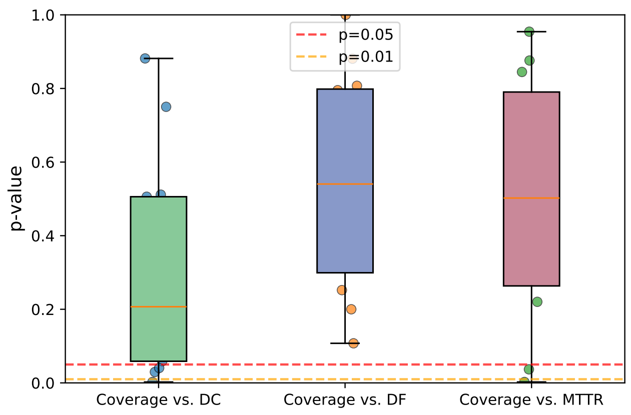

Figure 4: Box plot of p-values between CI coverage and DevOps metrics over all projects.

6 Discussion

The ’necessity’ of frequent execution As stated earlier in Observation 1 and 2, most of the times, workflows are ran on pull requests or pushes. 13.24% of all executions were triggered by a push and 75.47% of them were successful. A phenomenon that may sometimes occur is the running of tests on pushes to branches intended for feature development or bug fix (i.e. branches other than `main` or `master`. The results for these runs won’t be important as only the final run before merging will determine if this pull request is to be accepted.

For example, consider a pull request that has some commits before it is ready for merging which were uploaded to GitHub using n pushes. This would result in n workflow executions. If then the developers make another m commits to fix the pipeline (where $m \geq 0$), this will result in a maximum of $n + m$ executions, whilst the final one will decide if the request is to be merged or not. If the workflow is ran only for when the pull request is ready for merging, this will already reduce the executions by n.

This is already the case in many of the analysed workflows - they execute only on pull requests or pushes to their main branch. Thus, they isolate their CI pipelines from other branches reducing the number of unnecessary executions. For the project `keycloak/keycloak`, CI tests were even put to run on a scheduled basis instead on each commit in order to reduce the load on the GitHub workflow runners [14].

Implications for researchers: Further research can be done into what are the causes behind manual CI workflow execution (not strictly focused on CIs that execute tests). We saw an example where the ”manual” executions were done by an automated bot, while in another case the pipeline was executing regression tests which, perhaps, required closer human inspection on each execution. In addition, `ISSUE_COMMENT` events triggered 215266 runs that aren’t skipped. Manual evaluation of each one of them or their source YAML files is out of scope of this paper, but as seen on Observation 2, people use such configuration to trigger CI workflows by comments. This is de facto manual execution of workflows.

As a suggestion for collecting information for such a study, one can follow these steps: After creating a dataset of sampled repositories, collecting their workflows and the triggers of the workflows is easy to implement (given that the workflow files must follow a specific YAML format to be considered by GitHub). One can then filter on workflows that have turned on the trigger for manual execution - `WORKFLOW_DISPATCH` or `ISSUE_COMMENT`. The repository identifier (owner and name) and the ID of the workflow is enough to collect the workflow runs for the last 400 days including what triggered them and their conclusion.

Another question that the current results pose is the one of the presence of multiple platforms on which coverage is measured. Given the difference in the coverage on the different configurations for the `nodejs/node` project's CI workflows, it is reasonable to believe that there is benefit to run and test projects on different architectures or host operating systems. Two projects: `dbt-labs/dbt-core` and `modin-project/modin` also on different versions of python, while `doctrine/dbal` (a database abstraction layer software) on many different database providers - MariaDB, PostgreSQL. We suggest further research in the benefits of multi-architectural, multi-tool, multi-version and multi-OS running CI tools and its benefits and drawbacks. Interesting insights can be collected on which kinds of projects see better performance with different configurations (e.g. low-level parallelised software may benefit more when testing on many CPU architectures).

Further the relation between code coverage and DevOps metrics can be analysed using more sophisticated models for presence of non-linear correlation.

Implications for practitioners: Given the observed lack of statistical significance between test frequency and shifts in the selected DevOp metrics, we suggest limiting the number of CI workflow test runs. This can save on repository/team resources and, from an ethical stand point, limit consumption of computational power and electricity. Developers can instead choose to run the CI on GitHub only when a pull request is ready for merging and instead run their verification scripts locally.

Threats to Validity: Detecting if a Workflow execute tests is done by checking if any step either has a test-suggesting keyword in its name or if it runs a command which matches on a predefined regular expression. However, in certain cases this may miss custom scripts that execute tests (e.g. `./unit-tests.sh`) or non-standard runners.

Further, Given the human-controlled nature of issue management and the way we automatically detect issues as ones that report bugs, there is a chance some of them are misclassified. As explained in the methodology section, we classify issues as bug reports if they include a bug-indicating keyword in one of the issue's labels. The mitigation strategy we used was to filter only on projects that actively use issues and actively use labels (at least 100 issues where at least 75% are labelled). However, this may still allow for false positives or false negatives, e.g. a bug reporting issue left unlabelled or incorrectly labelled, an issue wrongfully labelled as bug or perhaps typos in the spelling of the label name or description.

7 Responsible Research

In this section we will discuss the ethics of this study and our approach.

Reproducibility The tool for collecting the data is available on Zenodo.org under 10.5281/zenodo.15711349. The GitHub actions runs and logs are available for 400 days and 90 days respectively, unless configured otherwise by the maintainers of a repository. Fundamentally, rerunning the tool to collect data will produce a different result then the data we have. Therefore, we also provide the datasets we used throughout this project on Zenodo.org under 10.5281/zenodo.15715218.

Usage of AI LLM models AI was used in this project for suggesting refinements to the statistical analysis of the correlation between Testing frequency and the measurements in the DevOps metrics. Further, AI was used to debug and solve issues that arose when writing CI-tool-du, writing small-scale scripts for organising or processing collected data (e.g. removing empty lines in CSV files). AI was also used for debugging LaTeX syntax when rendering. However, when it comes to writing the paper, no AI was used for rewriting, rephrasing, editing or generating text or punctuation. All text written in this document is written and edited so without AI.

Positive ethical implications On another note, the result of this project arguably have indirect positive ethical effects. With the limitation of unnecessary CI execution, companies can save up on computational power which in turn will result in less electricity usage. Our observations show that having a responsible approach does not necessarily come at the expense of project performance.

8 Conclusion

In this paper we researched the connection between test executions in CI and open-source projects performance in terms of DevOps metrics, as well as investigating the usage of such GitHub CI workflows. We evaluated 476 projects, 1680 test-executing workflows and 5778608 total runs. Our study finds that over 69.48% of all executions were triggered by pushes or pull requests. When it comes to manually triggering executions, 859 out of the 1680 workflows had this option enabled and in some cases, certain workflows were entirely only executed manually. We did not find any statistical significance between frequency of test executions or CI code coverage and our analysed DevOp metrics - Defect Count, Delivery Frequency and Mean Time to Recovery.

Further, we explored many interesting practices used across the open-source community for executing tests in the CI. Our work demonstrates that resources can be saved by lowering the frequency of test executions on the CI, while also not hurting a project's performance. We suggest further work in analysing manually triggered workflows - when they appear and what effect does this feature have.

References

[1] Iso/iec/ieee international standard - systems and software engineering–vocabulary. *ISO/IEC/IEEE 24765:2017(E)*, pages 1–541, 2017.

[2] Kent Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[3] Moritz Beller, Georgios Gousios, and Andy Zaidman. 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)*, pages 356–367, 2017.

[4] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[5] Omar Elazhary, Colin Werner, Ze Shi Li, Derek Lowlind, Neil A Ernst, and Margaret-Anne Storey. Uncovering the benefits and challenges of continuous integration practices. *IEEE Transactions on Software Engineering*, 48(7):2570–2583, 2021.

[6] Martin Fowler and Matthew Foemmel. Continuous integration, 2006.

[7] Google Cloud. 2024 dora accelerate state of devops report. https://cloud.google.com/devops/state-of-devops, 2024. Accessed: 2025-05-12.

[8] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 392–401, 2013.

[9] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 426–437, 2016.

[10] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[11] JetBrains. 12 benefits of ci/cd. https://www.jetbrains.com/teamcity/ci-cd-guide/benefits-of-ci-cd//. Accessed: 2025-05-12. Undated.

[12] JetBrains. Measure ci/cd performance with devops metrics. https://www.jetbrains.com/teamcity/ci-cd-guide/devops-ci-cd-metrics/. Accessed: 2025-05-12. Undated.

[13] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering*, 21:2035–2071, 2016. Publisher: Springer.

[14] Keycloak. Run testing workflows on a nightly basis instead of push · issue 10913 · keycloak/keycloak.

[15] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 155–165, 2014.

[16] Gustavo Sizilio Nery, Daniel Alencar da Costa, and Uirá Kulesza. An Empirical Study of the Relationship between Continuous Integration and Test Code Evolution. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 426–436, 2019.

[17] Eliezio Soares, Gustavo Sizilio, Jadson Santos, Daniel Alencar Da Costa, and Uirá Kulesza. The effects of continuous integration on software development: a systematic literature review. *Empirical Software Engineering*, 27(3):78, 2022.

[18] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. page 805 – 816, 2015. Cited by: 313.

[19] Fiorella Zampetti, Gabriele Bavota, Gerardo Canfora, and Massimiliano Di Penta. A Study on the Interplay between Pull Request Review and Continuous Integration Builds. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 38–48, 2019.