Exploring Descriptive Metrics of Build Performance: A Study of GitHub Actions in Continuous Integration Projects

Radu Stefan Constantinescu

Supervisor(s): Sebastian Proksch, Shujun Huang

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Radu Stefan Constantinescu
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Shujun Huang, Fenia Aivaloglou

Abstract

The Continuous Integration (CI) practice, has been rapidly growing and developing ever since it's introduction. This practice has been constantly providing benefits to developers such as early bug detection and feedback to development teams. In this study, we aim to identify the descriptive metrics that best illustrate the performance of the CI build stage, regarded as heart of the development process.

We conduct a small case study on repositories utilizing GitHub Actions, a CI tool that is relatively unexplored. Within this context, we classify projects using two performance indicators: build breakages and build durations. We examine two distinct sets of metrics in our analysis. The first set being build level metrics, which are closely linked to the build stage. The second set including project level metrics.

Our findings suggest that patterns traditionally associated with low breakages and durations are applicable to repositories employing GitHub Actions. However, understanding the relationship between project level metrics demands a more comprehensive approach, necessitating a thorough analysis of the project context for a holistic understanding of build performance.

1 Introduction

Continuous integration (CI) is a widely recognized practice in software development, involving the frequent merging of minor modifications into a shared central repository [1]. This practice, when adopted, comes with numerous benefits such as early feedback on code alterations and early detection of bugs in the development cycle, drastically reducing risks for development teams [2], [3]. Over time, the concept of CI has seen substantial evolution and refinement.

A significant milestone for Continuous Integration (CI) was established in 2006 when Fowler and Foemmel [2] outlined ten fundamental CI practices. These were intended to serve as a guide to enhance software quality and expedite the development process. After the introduction of these explicit steps, they were swiftly adopted and incorporated within the industry. Notably, immediate enhancements in software quality [1] and software project team productivity [4] were observed.

Up until this time, there exist studies [1], [4] that focus on CI improvements, with some of them raising the concern that context and project constraints might influence the way CI is to be implemented across different organizations [1], [5]. This concern raised the idea that there might not exist a ”one size fits all” solution for the CI implementation in different project contexts and further research has to be done in this area.

In this sense, the knowledge gap we want to address is: What are the contextual descriptive factors of software projects that affect the most the implementation of CI practice, and how can these factors be united in order to provide a guideline for maturing the continuous integration process.

As Continuous integration has been a subject of extensive research for several years it is clear that studying CI is a complex matter that requires multi-dimensional approaches. To obtain this holistic view, our research team narrowed down our analysis into project activity, maturity and topic, and the implementation of the CI build stage and pipeline.

In this paper, we will focus on the **build stage**, also regarded as the ”heart of software development ecosystem” [6], a critical phase, as in the software development process, most stakeholders get to interact with it [7].

Given its importance we can see how problems such as extended build times, failures, and frequent breakages not only disrupt this stage but can also ripple across the entire development work [8], having serious effects on programmer's productivity [9].

Building upon these observations, our study aims to unravel: **What are the most descriptive metrics for identifying build performance?**

In order to answer the main question of this paper, we will break it down into two sub-questions.

- **RQ1:** What are the key **build level metrics** that significantly contribute to the evaluation of build performance?
- **RQ2:** What are the essential **project level metrics** that play a significant role in the assessment of build performance?

In the subsequent sections of this paper, we will dive into pertinent research, spotlighting a handful of notable studies that have informed our methodology. For a smooth understanding, we shortly explain the structure employed by Github Actions, then we present an in-depth step exploration of our data collection process. We will then transition to examine the findings pertaining to our research questions, followed by a comprehensive discussion regarding the discoveries made and the limitations and threats to validity of our findings.

2 Previous work

In software development, particularly when viewed through the lens of Continuous Integration (CI), the concept of build performance is multifaceted and subjectively valued. The unique concerns of various stakeholders significantly influence the interpretation of this concept. For some, the discourse may pivot around financial considerations, as detailed in a paper focusing on cost-effective CI strategies [3]. Others might prioritize the quality of the software process, emphasizing the efficient and error free builds [4].

In exploring the field of CI build performance, a substantial body of research is devoted to the phase of building, with a particular emphasis on examining the diverse nature of build failures. Several studies concentrate on uncovering the reasons behind build failures [10], [11], [12], while others, approaching the issue from a unique angle, finding patterns of failures with the goal of predicting build outcomes. Their aim is skipping non-failing builds, in order to reduce the overall cost of the CI stage [3].

On a similar note, the duration of the build process represents another key performance metric that frequently becomes a focal point in CI improvement research. Some research targets the reasons contributing to extended build times, investigating factors that may cause unnecessary delays in the build process [13]. Simultaneously, research pinpointed anti-patterns in CI processes, implying that slow builds might be a damaging practice in software development [14]. Meanwhile, other explore the impact of lengthy, resource-draining builds on team behavior, highlighting their substantial effect on team interactions and productivity [15].

While the metrics of build failures and build durations receive substantial focus in research on CI build performance, it is important to understand that these metrics are often explored in isolation, potentially leading to a fragmented understanding of build performance [5].

The study of Ghaleb et al [5] aims to bridge this gap. Instead of looking into these performance aspects in isolation, they decide to study build durations and build breakages and the way in which they are connected. By expanding the TravisTorrent dataset [16] with additional Travis CI [1] projects and conducting an in-depth analysis of over 900,000 builds, they identify which factors are most tightly correlated with desirable performance. Moreover, they validate their results by conducting developer surveys. They look into which build level metrics and project level metrics have a significant impact on build performance, and find that often trade offs have to be made, and one performance aspect has to be sacrificed in favour of another.

With a similar methodology, considerable portion of current research surrounding the build stage and CI consistently utilizes Travis CI [3], [11] along with the TravisTorrent dataset, [16]. The dataset, assembled from over 1000 projects utilizing Travis CI and mined from GitHub repositories in 2017, was created to simplify the study of CI for researchers. This methodology has gradually broadened our understanding of the CI landscape that leverages Travis CI. However, despite Travis CI's significant role in the CI field, numerous other tools exist that demand a more in-depth exploration, as underscored by recent build performance studies [5].

In this sense, we decide to shift our focus to GitHub Actions, a new entrant in the CI tools sector that remarkably surpassed Travis CI, the longstanding market leader, within 18 months of its launch [17]. Our research will extend the analysis to multiple branches beyond just the master/main, as suggested for future research in [5] and will study the performance by looking at build breakages and durations together by exploring various project and build level metrics.

3 Github Actions

Introduced in November 2019, GitHub Actions (GHA) is a new player in the CI landscape, but has rapidly ascended to a dominant position [18]. The service's swift growth can be attributed to its seamless integration with GitHub's ecosystem and a flexible, robust workflow configuration, making it a compelling option for developers [19].

GitHub's (GH) prominence as the primary social coding platform has significantly accelerated GHA's adoption. With a vast user base of over 94 million users in 2022, including 414 million contributors and use by 90% of Fortune 100 companies [20], GHA is in a unique position in the CI landscape.

To employ GHA, developers are required to configure a workflow file with a YAML extension, which determines the structure of the workflow. Each workflow revolves around specific events. When these events occur, they trigger the execution of jobs. Each job is composed of several steps, which are undertaken sequentially [21].

Even though this sequence of events, jobs, and steps adheres to a standardized structure, GHA provides a significant level of customization at each stage, offering considerable freedom to developers. They can modify triggering events, define the specifics of jobs and steps, select runners, manage job dependencies, and control the workflow's behavior upon the failure of individual steps. This adaptability allows developers to tailor any time of workflow automation to meet their project's unique needs.

Listing 1 demonstrates an example of a GitHub-defined workflow template for the "build and test" stage, illustrating a configuration of a Java Maven Build stage in GHA [22].

Listing 1: Build & Test Java with Maven

```
name: Java CI
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up JDK 17
        uses: actions/setup-java@v3
        with:
          java-version: '17'
          distribution: 'temurin'
      - name: Build with Maven
        run: mvn --batch-mode --update-
            snapshots package
```

4 Methodology

For this study, we opted for Python [2] due to its flexibility and the availability of numerous libraries that enable easy access to APIs and data processing. Central to our tool's functionality was PyGitHub[3], a client-side library that facilitated our tool's end-to-end processes, ranging from the filtering stage to the extraction of repository-related data. However, there were instances when GitHub's API[4] offered features not available in this library, such as interaction with GHA. In these situations, we resorted to raw HTTP requests for data extraction, as in the case of extracting jobs from a given workflow run.

We constructed our tool using a pipeline stage architecture, promoting enhanced re-usability and methodical progression through the extraction stages. The structure can be categorized into three primary divisions: Repository Generation, Data Extraction, and Data Analysis, each hosting a variety of specific sub-stages tailored to their respective functions.

[1]https://www.travis-ci.com

[2]https://www.python.org

[3]https://pygithub.readthedocs.io/en/latest/

[4]https://docs.github.com/en/rest/actions/workflow-jobs?apiVersion=2022-11-28

Table 1: Build Level Metrics

Category	Metric	Description
Configuration	Cache usage	Usage of GHA caching actions in the build configuration
	Fail Fast Usage	Usage of the Fail Fast GHA option in the build configuration
	Skip Usage	Presence of the skipping builds practice
	Job Count	Number of jobs configured per GHA workflow
	Job Churn	Added Jobs - Deleted Jobs / consecutive build
Failure	Q4 Breakage Rate	Frequency of failure conclusions divided by overall conclusions count in the last 25% of the project's lifetime (q4)
	Retrial Breakage Rate	The breakage rate calculated by grouping build runs according to their run_-attempt attribute
	# of consecutive breakages	The number of failures occurring from the first failure event to the next successful one
Duration	Build duration	The time difference between when the GHA Build Workflow run ended and it started
	Recovery Time (min)	The amount of time it took for developers to fix a failed build
CI Activity	Weekend/Weekday Build	Whether the build has been triggered during a weekday or weekend
	Peak week day	Day with the most CI activity

Table 2: Project Level Metrics

Category	Metric	Description
Code Maturity	Project Age in years	Repository age as identified from the created_at GitHub attribute
	Project Size (KB)	Repository size in KB
Activity	# Commits	Number of commits in the lifetime of the repository
	# Branches	Number of active branches in the repository
	# Releases	Number of releases in lifetime of the repository
	# Contributors	Number of contributors in lifetime of the repository
Reputation	# Stars	Number of the stars
	# Forks	Number of the forks

Furthermore, stage yeild intermediate outputs, and present configuration files for flexibility. This configuration file provides stage-specific settings, ensuring the tool's adaptability to different inputs, making it highly customizable and versatile. An example of such a configuration can be found in Listing 2, where we outline the parameters for the Repository Generator stage.

Listing 2: Repository Generator Configuration

```
"RepoGenerator": {
"languages": ["JavaScript", "java"],
"repo_number": 400,
"min_stars": 75,
"min_days_old": 365,
"min_contributors" : 5,
"min_commits" : 100,
}
```

4.1 Repository Selection

In our research, we utilized several filters to compile a representative set of repositories, excluding 'dummy' and 'toy' projects in line with prior studies [23], [3]. We find repositories by querying the GitHub API using the search functionality. In selecting repositories we used two set of of filters: project level and CI level. We will first discuss the project-level filters criteria.

1. **Repository Activity:** We asses the lifespan activity for a repository with a threshold of a minimum 100 commits.
2. **Popularity:** We further established a popularity criterion, demanding each repository to have garnered at least 100 stars, indicative of significant community interest or endorsement.
3. **Established Date:** We enforced a criterion for the repository's age, including only those repositories that were created at least a year prior, ensuring a sufficient time-frame for maturity and the likely stabilization of their CI setup.
4. **Team Size:** We established a minimum requirement of at least five contributors for each repository, enabling us to consider the impact of collaboration on CI practices.
5. **Non Forked** Furthermore, to maintain the authenticity of our dataset, we excluded forked repositories or those that were merely copies of other projects.

In terms of CI level filtering, we aim to identify the build stage out of all repositories workflows, a task that entails complexity due to the inherent flexibility of GHA workflows [21]. We take the following steps in order to ease our filtering task.

Primary Programming Language We chose to limit our search to three primary programming languages: Java, Type-

Script, and JavaScript. This choice is further justified by the fact that these languages currently rank among the top 4 most popular on GitHub [20].

Workflow Analysis Once our search space was refined, we initiated an in-depth analysis of the workflows of our 400 selected repositories. This analysis focused on understanding different aspects such as titles (of the workflow file, jobs, and steps), GHA actions used and scripts executed within steps. We download all workflows locally making a total of 1745 workflows from 264 repositories.

Scoring System: We devised a scoring system. We award points based on the following criteria. We look for keywords (i.e) (”CI”, ”Release”, ”Build”, ”Test”) for the names and titles of workflows jobs and steps. We look into the presence of common build commands and actions, and we award extra points for common stages that typically occur after the build stage. We do this by analyzing all workflow builds and by taking inspiration from GH docs [24].

Template Inclusion Based on previous studies, [17], we understand that more than 50% of developers make use of configurable templates provided by Github when creating their workflows. In this sense we run the scoring on the templates associated with our choice of programming languages, and we use them as a threshold to ensure that even basic configurations can pass the filtering.

Workflow Scoring Filtering Post-ranking all workflows, we initiated a series of statistical filters for further investigation. Initially, we normalized the scores based on the amount of times a workflow's score was increased, in order to prevent long configuration files from accumulating exaggerated scores. We look into empirical studies, where they indicate that most GHA workflows define 1-2 jobs [19], so we expected simpler workflows to score lower, while more complex ones could score higher due to their intricate structure. This resulted in a multitude of workflows achieving a score of zero, 1102 (63%) and being filtered out for efficiency, and the filtering of 35 repositories, accounting to 13% of repositories.

In the subsequent step, we applied the Interquartile Range (IQR) method to pinpoint and eliminate outliers from our data. The IQR method excels in handling extreme values by centering on the middle 50% of scores, enabling us to exclude data points that significantly deviate from the average. Post outlier removal, we computed the average workflow score and used this value as a benchmark. Workflows scoring below this average were dismissed.

After applying the filtering and scoring mechanisms, we obtained a final set of **66 repositories and 84 workflows,** excluding a total of 1,213 workflows, starting from 300 repositories. Notably, our dataset includes the repository ”GitHubTemplates” that we created, incorporating the templates provided in the GitHub documentation. This inclusion ensures that even simple templates are encompassed by our workflow filtering process.

4.2 Performance Clustering

In our aim to categorize our selected repositories based on the build performance, it is essential to conduct a thorough examination of a significant number of builds within each workflow. Drawing from the insights in Ghaleb et al.'s study [5], their sensitivity analysis suggests that analyzing the entirety of build history might not be viable. Therefore, we have opted to use a similar strategy by segmenting projects into lifespan quarters for the purpose of categorization. Consequently, we strive to extract, if feasible, only the last 25% of the total workflow runs, representing the most recent quarter. If this proves unattainable, our extraction will be limited to the final 1000 runs.

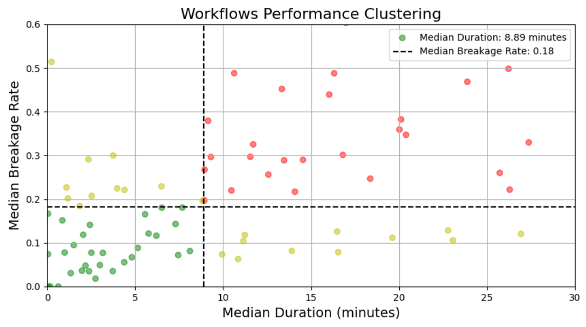

Figure 1: Workflow Performance Clustering

This choice to focus on the last quarter of builds was influenced by various considerations. These include the time restrictions of our study, the limit of 5000 requests per hour set by GitHub's API [25], and our performance clustering decision, which was informed by the finding that ”80% of projects maintain their build performance for more than three quarters of their lifetime” [5].

In order to find out the build performance of the selected repositories, we use two build performance metrics: build breakage and build duration. For each workflow, we calculate the median build duration and their breakage ration, as the amount of ”failure” conclusions / failures and successes, we omit the other conclusions, as some workflows may be intentionally omitted/skipped.

We then use the computed medians, median breakage ration (18.9%), Figure 1, and median build duration (8.89 minutes), Figure 2. We then classified the projects into quadrants, following our reference methodology [5] in order to correctly identify the project's current state.

We identify 4 quadrants based on these 2 splits.

1. The bottom-left quadrant, Low Breakage/Low Duration (LBLD) indicating projects in which the majority of the analyzed builds have low breakage rate and low duration. We identify 30 workflows falling in this category.
2. The top-left quadrant, High Breakage/Low Duration (HBLD) indicating projects in which the majority of the analyzed builds have high breakage rate and low duration. We identify 12 workflows falling in this category.
3. The bottom-right quadrant, Low Breakage/High Duration (LBHD) indicating projects in which the majority of the analyzed builds have low breakage rate and low duration. We identify 12 workflows falling in this category.
4. The top-right quadrant, High Breakage/High Duration (LBLD) indicating projects in which the majority of the

analyzed builds have high breakage rate and high duration. We identify 30 workflows falling in this category.

4.3 Metric Selection

Upon reviewing the literature, it's evident that research typically bifurcates the data into two key categories: a high-level category referred to as project level metrics, and a low-level metrics related to the build stage referred to as build level metrics [5], [3].

These research efforts typically begin with an exhaustive list of metrics, and through analysis, distill down to the most relevant ones. Following this approach, we will detail the metrics that we've chosen to examine from past research for each of these two categories project and build level metrics in the following sections.

Build Level Metrics

Build Level Metrics are metrics that look into various specifics of the build stage such as details regarding low level configurations such as: caching, fail-fast, the use of retries. These metrics are all closer to the build stage and when analyzed reveal valuable insights on desirable patterns when it comes to build performance.

The exhaustive dataset provided by TravisTorrent [16] offers a wealth of metrics that can be highly beneficial in analyzing the build stage. By integrating these metrics with the insights from Ghaleb et al's study, we focus our attention on the metrics or actions identified most relevant in reducing build times and ensuring successful builds [5]. Given that their study concentrated on Travis CI, we aim to identify analogous configurations present in GHA. A description of a subset of the metrics can be seen below:

1. **Caching** In order to detect cache usage in a workflow, we referred to the public documentation of GitHub, which provided us with the insight, that to enable caching on dependencies or jobs, either a caching action can be used or using the corresponding package managers setup actions that take care of caching for you [26]. A few examples of these are Gradle and Maven with setup-java action, or npm, Yarn, and pnpm with setup-node. With this understanding, we analyzed the actions used in each workflow, specifically seeking out those that involve caching, regardless of version. We took into account setup actions for all package managers that we discovered in our workflows. Given the restrictive decision of three programming languages for this study, this approach allowed us for consistent detection of caching usage in a GHA workflow.
2. **Fail-fast Usage** Another significant property in the build configuration is the fast-finish option in Travis CI. This option allows when enabled for immediate build result after all required jobs have been finished [27]. A similar setting exists in the context of GHA, Fail-fast, with similar functionality, it enables a "fast failure" if any of the matrix jobs fails [28]. We look into the frequency in which this configuration appears in the workflows, with the note that by default for GHA, the configuration is enabled.
3. **Retry Performance** Another metric which was hinted as a possible way of improving build performance is the retrial times of the builds. In this sense we look for the breakage of workflows filtered by their run_attempt GHA workflow run attribute.
4. **Resolution Time** In addition to metrics analyzed by Ghaleb et al's work [5], we also "resolution_time", inspired from by a case study at Google, [10], where they conduct an empirical study on programmer's build errors. Resolution time is described as the time it takes between the first occurrence of a failure and the next build event which concluded into a successful build. It aims to calculate the time a developers takes to fix a broken configuration.
5. **Recovery Attempts** Additionally, in a similar fashion to Resolution Time we looked at the Recovery Attempts, which were calculated by the amount of failures between the first failure and next success.
6. **Skip Usage** We looked all conclusions extracted of each workflow and we identified whether they had a skip conclusion in their skip conclusion sets.
7. **Weekday/Weekend** We looked into the amount of breakages that occur during weekdays and weekends, and the most active CI day to see if there are any patterns emerging.

The above described metrics along with additional ones can be found in Table 1, Build-Level Metrics.

Project Level Metrics

Project-Level Metrics embody the high-level features of a repository, such as code activity, project maturity and reputation. These metrics offer valuable insight into aspects such as the maturity of a project, its reputation in the developer community, its rate of evolution, activity, and can be viewed as a preface to a project's face. Previous work [5] shows that these types of metrics indicate a high association with the two build performance aspects studied: build breakage and build duration.

Table 2 encompasses the project level metrics that have been selected.

5 Experimental Setup and Results

5.1 RQ1: What are the key build level metrics that significantly contribute to the evaluation of build performance?

From analyzing the above-mentioned build level metrics we identify the following observations:

1. **Repositories test diverse configurations more frequently on non-primary branches.** Table 3 illustrates the branch usage variances among clustered projects. Regardless of the cluster, primary branches consistently display lower breakage rates than their development counterparts. Notably, clusters with fewer breakages, such as LBLD and LBHD, exhibit a substantial difference in breakage rates across branch types (a threefold and twofold difference respectively). However, projects

associated with high breakages show comparably similar rates across branches.

2. **We identify the principle of "if it's not broken don't fix it" among studied projects.** As hinted in Ghaleb et al's. work [5], the principle of "never change a winning horse" seem to be present among clusters. Our calculations of job churn across successive workflows typically reveal a near-zero percentage (see Table 3). However, an exception arises within the HDHD quadrant, marked by a 0.27 value. This implies that repositories in this category might be recognizing and addressing their performance issues.

3. **Keeping the build configuration simple and solving failures fast is associated with overall good performance.** Research on Github Actions usage [19] suggests that most workflows are configured with just one job. We observe that, with the exception of the HBHD cluster (which has a median of five), all other clusters configure only a single workflow on their median. For these clusters, their 80th percentile reaches a maximum of two configurations, whereas it spikes to nine for the HBHD cluster (see Table 4). Moreover, we see how on the main branch, there is a clear increase in resolution time between clusters as we go further away from LBLD, observation in line with Fowler's "Fix Broken Builds Immediately" principle [2].

Table 4: Stats of # Jobs Configured / Workflow

Quadrant	Average	Median	p80	p95	p99
LBLD	1.50	1.00	2.00	3.55	5.42
LBHD	1.94	1.00	2.00	5.00	7.4
HBLD	1.08	1.00	1.00	1.45	1.89
HBHD	6.91	5.50	9.00	18.05	29.03

4. **As run attempts of a workflow grow the more likely the results conclude in a failure.** Figure 2 shows a clear escalation in breakage rates as the number of attempts rises across all clusters. Notably, the HBHD cluster exhibits a relatively consistent distribution of breakages per run attempt, suggesting a potential prevalent practice of retrial within this group. This trend is less apparent in the other clusters, where the number of breakages decreases (Table 5). This observation is in line with the developer's perception, which state that "retrying commands is not always helpful in fixing build breakage and should rather be a last resort", opinion confirmed by surveying on the opinion of retrying failed builds [5].

5. **CI build activity might be dependent on project context.** We explored the relationship between performance of builds distinguishing between weekdays and weekends. Table 7 reveals consistent performance patterns across these groups. However, the ratio of weekday to weekend builds varies substantially across clusters, with low breakage clusters exhibiting a 6.2-1 ratio, compared to a 3.05-1 and 4.78-1 ratio for the high breakage quadrants. This variation might reflect differences in project

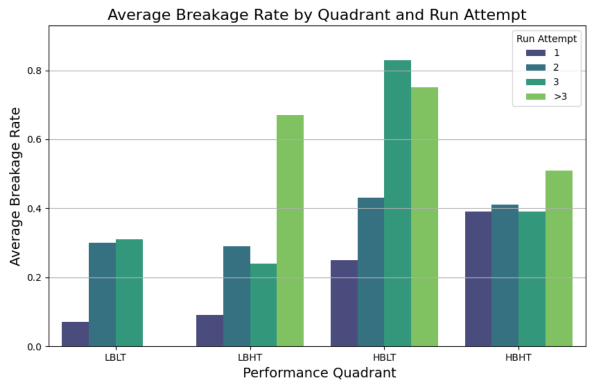

Figure 2: Performance on Retrial Attempt

practices or contexts. Furthermore, our heat-map of frequency of peak weekly activity shows Thursday as the most active day across all clusters (Figure 3). Notably, for low breakage clusters, the early weekdays - Monday and Tuesday - seem to be rather active than the counterparts. This observation suggests potential organizational differences among clusters, possibly related to whether they are company-managed or open-source projects.

6. **Caching, Fail-Fast and skipping builds are closely coupled to one build performance aspect.** Table 6 provides insights into the relationship between various build configurations and performance clusters. The key takeaway here is that each configuration metric appears to align with a specific performance metric.

   In the case of the fail-fast configuration, it is less frequently disabled in low breakage clusters (with only 6.25% and 6.67% disabling), whereas high breakage clusters show a contrasting trend, with the configuration being disabled more often (16.67% and 26.47%), pattern observed in previous studies as well [13].

   Cache usage shows similar patterns. It is prominently active in workflows with low durations, signifying its role in efficient build processes. However, as one might anticipate, high duration workflows tend to disable caching more frequently, possibly due to the potential overheads or the nature of tasks involved.

   When examining the skip command, it's noteworthy that its usage is nearly absent in high breakage scenarios (0% and 2.94%), suggesting that all steps are typically executed despite high failure rates. In contrast, low breakage clusters show higher instances of skipping builds, with one cluster indicating this practice in a quarter of its build configurations. These observations underline the strategic decisions taken by repositories within different performance clusters, hinting that different practices might be adopted, depending on the value associated to build performance and possibly context of the project.

Table 3: Build-Level Metrics on different branches

Quadrant	branch type	breakage rate mean	resolution time median	consecutive fails mean	job churn mean	run count mean)
LBLD	main	0.05	47.10	0.25	-0.00	312.21
LBLD	others	0.15	0.00	0.45	0.00	9.60
LBHD	main	0.06	135.39	0.92	-0.00	235.44
LBHD	others	0.12	0.00	0.48	0.06	11.00
HBLD	main	0.25	92.38	6.53	-0.05	141.42
HBLD	others	0.31	5.76	1.00	-0.01	11.43
HBHD	main	0.33	219.98	13.91	0.00	235.64
HBHD	others	0.34	31.78	1.06	0.27	10.38

Table 5: Breakage - Retrial Correlation

Quadrant	#Attempt	% Breakage	# Breakage
LBLD	1	0.07	30
	2	0.30	21
	3	0.31	9
	>3	0.00	0
LBHD	1	0.09	16
	2	0.29	12
	3	0.24	8
	>3	0.67	3
HBLD	1	0.25	12
	2	0.43	11
	3	0.83	3
	>3	0.75	5
HBHD	1	0.39	34
	3	0.39	25
	2	0.41	33
	>3	0.51	30

Table 6: Fail-fast (FF) and No Cache and Skip Usage in Performance Clusters

Quadrant	Fail Fast Disabled %	Cache Disabled %	Skip Usage %
LBLD	6.67%	6.67%	13.33%
LBHD	6.25%	12.50%	25.00%
HBLD	16.67%	0.00%	0.00%
HBHD	26.47%	14.71%	2.94%

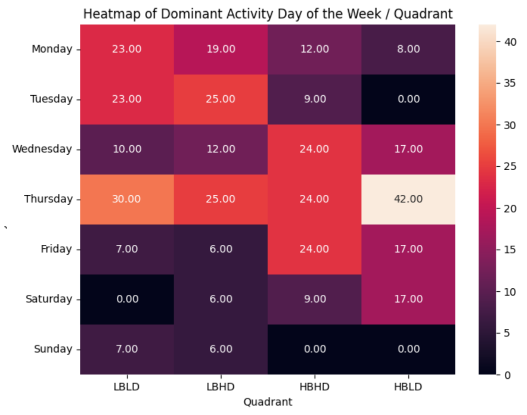

Figure 3: Heatmap of Dominant Activity in Day of the Week

Table 7: Day Type Performance

Quadrant	Day Type	Total	Failures	Failure Rate
LBLD	weekdays	4247	265	0.062397
	weekend	696	35	0.050287
LBHD	weekdays	2518	260	0.103257
	weekend	402	35	0.087065
HBLD	weekdays	4120	1040	0.252427
	weekend	1348	383	0.284125
HBHD	weekdays	6247	1845	0.295342
	weekend	1324	384	0.290030

5.2 RQ2: What are the essential project level metrics that play a significant role in the assessment of build performance?

We used parallel coordinates plotting to illustrate the patterns derived from the performance clustering classification. This method effectively represents multidimensional data, such as multiple numerical project-level metrics we are examining concurrently. To make these diverse metrics comparable, we normalized their values. (See Figure 4)

We make the following observations:

1. **Keeping the team size low and having a small project is associated with good performance.** We see that the desired cluster, (LBLD) has the least number of contributors and also the smallest project in terms of size. We also observe that being not as popular (stars and forks) as the others projects, they seem to have a high number of branches, hinting towards multiple best practices such as avoiding pushing to mainline and implementing small features on feature branches.

2. **Popularity might pressure into poor performance, when certain maturity level has been reached.** Clusters with predominantly long durations and frequent breakages (HBHD), present the most number of stars, contributors and a high maturity level (age and size). This mix of observations suggest that the raised number of contributors could be a response to the increase

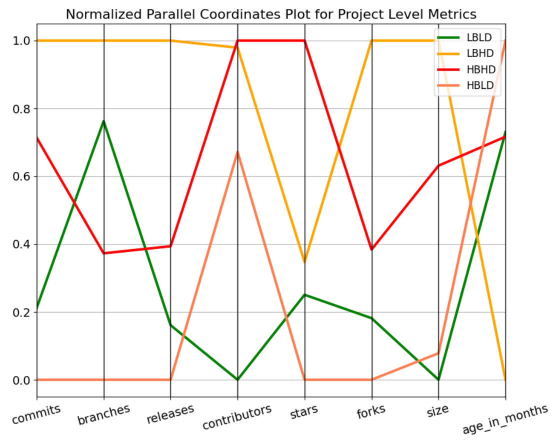

Figure 4: Normalized Project Level Metrics across Performance Quadrants

in complexity, hence the high maturity levels, as more work is required in order to advance the development. This can however come at the cost of performance, due to difficulties in coordinating increased teams and managing complex codebases. Moreover, the maturity factor could indicate that either these projects have met with a unexpected popularity and adoption of best practices was not a top priority, or that these projects have grown so complex that adoption of best practices became a challenging task.

3. **Maturity strongly influences build performance, possibly forcing trade offs between performance aspects** For the 2 clusters, LBHD and HBLD, which seem to be in a favourable state with only one of the two performance metrics, we identify opposed maturity characteristics. In the case of HBLD, we observe how they are the most mature in terms of age but have a simple project, deduced from the size. Moreover they seem be the most unpopular among clusters, and with the least development, however with a relatively high number of contributors. Looking at LBHD, we identify opposite maturity aspects. Those projects seem to be the youngest however with the most complex projects. Interestingly despite intense activity (most commits, branches, releases), they seem to not be the most popular projects.

6 Responsible Research

This section discusses potential threats to study validity, our approaches to mitigate these, and the reproducibility of our methods. Adherence to ethical standards and commitment to reproducibility, underscored by transparency, objectivity, and accountability, have been foundational in our research.

6.1 Construct Validity

Construct validity concerns the degree to which our measurements truly reflect the constructs they are intended to represent. We detail potential challenges to our construct validity as follows:

- Despite our repository selection based on prior methodologies and the use of the GitHub API, its inherent bias towards displaying popular/active projects could potentially influence our data extraction and subsequently, our results.
- The scoring system we implemented, primarily focused on isolating build stage workflows, was principally an extraction tool, not a core focus of our research. Despite its emphasis on specific keywords and actions, it might overlook certain custom configurations. Recognizing the challenge of comprehending all intricate configurations and the knowledge gaps regarding GitHub Actions, we incorporated GitHub templates to ensure the inclusion of basic workflows.
- We recognize the possibility of results being skewed by inaccuracies in value computations.
- We admit that utilizing medians and means for visualizing data and clustering across performance quadrants may inadvertently obscure valuable information, leading to potential misinterpretations in findings.

6.2 Internal Validity

Internal validity refers to the extent to which the design and conduct of the study ensure that the findings are solely due to the variables the researcher intended to investigate, without influence from other confounding variables. In the context of our study, potential threats to internal validity are identified as follows:

- Our conclusions are primarily drawn from patterns observed by classifying build performance into quadrants, using methodologies from previous studies. These classifications, however, encompass only two recognized aspects of build performance.
- We acknowledge the limitation that our study only employs a subset of potentially important metrics for build performance at both the build and project levels. To address this, we analyzed metrics previously identified as important, yet our selection of project level metrics was constrained as other researchers in our group are focusing on them. We consciously chose not to dive into more complex metrics in this study.

6.3 External Validity

External validity refers to the generalization of our results to other settings and contexts. We have the following observations:

- We study a limited number of 66 repositories with 84 workflows. We include only workflows which have at least 500 runs, to be able to capture a realistic frame of performance.
- We choose to limit our study to 3 popular programming languages, in order to ease our comprehension of the GitHub actions workflows. We take in this sense 3 out of top 4 popular programming languages.

- The decision to evaluate build performance by examining only the last 25% of the projects' lifetime may offer a narrow view of the entire project lifetime. It restricts visualising the whole performance trends or significant changes that happened across earlier stages of a project. We validate this decision with the fact that most projects keep the same performance quadrant across their lifetime, and the limit imposed by GitHub's API of 5000 requests/hour [25], combined with the limited time frame of the research project.

6.4 Reproducibility

Our code, instrumental to our research, is publicly accessible on GitHub under the Descriptive-CI-Metrics repository [5]. This open-source approach invites peer scrutiny and enables fellow researchers to validate, reuse, or extend our work, fostering a spirit of collaboration and cumulative knowledge growth. It also significantly enhances the reproducibility of our methods.

In organizing our code, we have followed a pipeline-like structure. This approach is easy to follow and further enhances the reproducibility of our research. It ensures that each stage of our process is clear, logical, and easily replicated, providing others with a detailed flow of the work.

Our research outcomes are presented with complete honesty. All plots and graphs are genuine reflections of the data extracted from the preceding stages. If we have excluded any data, we have not only clearly mentioned it but also explained the rationale behind this decision. This policy ensures our results are a true representation of our work, and maintains the integrity of our research.

7 Conclusions and Future Work

In this paper, we conduct a small scale study on GitHub projects, with the aim of finding which are the most descriptive factors that affect the implementation of CI practice. In order to achieve this, we focus on the build stage, specifically on the aspect of performance. We study build performance literature and identify that often build performance aspects are studied independently. We identify a reference study which looks at the interplay between build duration and build breakages. We follow their methodology and cluster our projects into four performance quadrants, in order to find the most relevant metrics that can be descriptive when looking at build performance.

To provide a comprehensive view, we bifurcate the metrics into two categories. First, build level metrics, which are closely tied to the build stage (i.e configurations). Second, project level metrics that show characteristics of a repository, including code activity, project maturity, and reputation. This analysis is conducted in the context of GitHub Actions, a CI tool that, because of it's novelty has not been as as extensively studied.

Following our investigation, our study confirms that practices and principles already recognized and implemented in the CI industry, hold valid in the case of GitHub Actions.

[5]https://github.com/raduConstantinescu/Descriptive-CI-Metrics

Upon examining the build level metrics, we revealed several insightful observations. In this sense, we saw the principle of "never change a winning horse", where projects within favorable performance quadrants rarely altered their build configurations.

Additionally we can reconfirm the impact of configurations such as caching, adoption of the 'fail-fast' strategy, and skipping builds are strongly tied to aspects of build performance.

These observations not only validate commonly accepted principles but also highlight potential strategies for enhancing the efficiency and effectiveness of CI practices in projects using GitHub Actions.

Simultaneously, our exploration of project-level metrics validates certain patterns. For example, keeping the team and project size small has a positive correlation with favourable build performance. Furthermore, it appears that implementing optimal CI practices from the project's early stage, before it reaches a mature level, may be beneficial.

However, despite these insights, it's important to note that the collective analysis of project-level metrics underscores the necessity for understanding the specific context of a project for a more thorough comprehension.

In summary, our observations confirm that, even when investigating a relatively new tool as GitHub Actions, we identified patterns consistent with those seen in other CI tools. This leads to the suggestion that certain CI practices may be universally effective, regardless of the specific tool utilized. However, this hypothesis requires further validation. To this end, we encourage comprehensive, large-scale empirical studies of GitHub Actions with a specific focus on build performance. Importantly, this studies should not only validate the emerging patterns on a broader scale but also take into account the project context, recognizing its role in fully understanding and optimizing CI practices.

References

[1] 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.

[2] Martin Fowler. Continuous integration. *martinfowler.com*, 2006.

[3] Xianhao Jin and Francisco Servant. A cost-efficient approach to building in continuous integration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 13–25, 2020.

[4] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 805–816, 2015.

[5] Taher Ghaleb, Safwat Hassan, and Ying Zou. Studying the interplay between the durations and breakages of continuous integration builds. *IEEE Transactions on Software Engineering*, pages 1–21, 11 2022.

[6] Shane McIntosh, Bram Adams, Thanh HD Nguyen, Yasutaka Kamei, and Ahmed E Hassan. An empirical study of build maintenance effort. In *Proceedings of the 33rd international conference on software engineering*, pages 141–150, 2011.

[7] George Neville-Neil. Kode vicious permanence and change. *Commun. ACM*, 51:27–28, 12 2008.

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

[9] Henry Lieberman and Christopher Fry. https://dl.acm.org/doi/fullhtml/10.1145/223904.223969, 1995.

[10] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: A case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 724–734, New York, NY, USA, 2014. Association for Computing Machinery.

[11] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. A tale of ci build failures: An open source and a financial organization perspective. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 183–193, 2017.

[12] Taher Ahmed Ghaleb, Daniel Alencar da Costa, Ying Zou, and Ahmed E Hassan. Studying the impact of noises in build breakage data. *IEEE Transactions on Software Engineering*, 47(9):1998–2011, 2019.

[13] Taher Ahmed Ghaleb, Daniel Alencar Da Costa, and Ying Zou. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, 24:2102–2139, 2019.

[14] Carmine Vassallo, Sebastian Proksch, Harald C Gall, and Massimiliano Di Penta. Automated reporting of anti-patterns and decay in continuous integration. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 105–115. IEEE, 2019.

[15] Graham Brooks. Team pace keeping build times down. In *Agile 2008 Conference*, pages 294–297. IEEE, 2008.

[16] Moritz Beller, Georgios Gousios, and Andy Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 447–450. IEEE, 2017.

[17] Sk Golam Saroar and Maleknaz Nayebi. Developers' perception of github actions: A survey analysis. *arXiv preprint arXiv:2303.04084*, 2023.

[18] Mehdi Golzadeh, Alexandre Decan, and Tom Mens. On the rise and fall of ci services in github. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 662–672, 2022.

[19] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. On the use of github actions in software development repositories. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 235–245, 2022.

[20] GitHub. The state of the octoverse. https://octoverse.github.com/. Accessed June 4, 2023.

[21] Github. Understanding github actions. https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions. Accessed June 4, 2023.

[22] Github. Build and test java with gradle. https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-java-with-gradle#using-the-gradle-starter-workflow, 2023. Accessed June 21, 2023.

[23] Chen Zhang, Bihuan Chen, Junhao Hu, Xin Peng, and Wenyun Zhao. Buildsonic: Detecting and repairing performance-related configuration smells for continuous integration builds. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.

[24] GitHub. About continuous integration. https://docs.github.com/en/actions/automating-builds-and-tests/about-continuous-integration, 2023. Accessed June 6, 2023.

[25] GitHub. Github docs: Resources in the rest api. https://docs.github.com/en/rest/overview/resources-in-the-rest-apiVersion=2022-11-28, 2022. Accessed June 7, 2023.

[26] Github. Caching dependencies to speed up workflows. https://docs.github.com/en/actions/using-workflows/caching-dependencies-to-speed-up-workflows#using-the-cache-action, 2023. Accessed June 7, 2023.

[27] Travis CI. Fast finish. https://docs.travis-ci.com/user/customizing-the-build/#fast-finishing, 2023. Accessed June 12, 2023.

[28] GitHub. Workflow syntax for github actions. https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#jobsjob_idstrategyfail-fast, 2023. Accessed June 18, 2023.