Evolution of CI Pipeline Complexity: Impact on Build Performance

Kwangjin Lee[1]

Supervisor(s): Sebastian Proksch[1]**, Shujun Huang**[1]
Examiner: Marco Zuniga Zamalloa[2]

[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 21, 2025

Name of the student: Kwangjin Lee
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Shujun Huang

Abstract

Continuous Integration (CI) has become a fundamental practice in modern software development. Organizations increasingly adopt complex pipeline configurations to automate their build, test, and deployment processes. Well-optimized CI/CD pipelines offer significant benefits in deployment reliability, team productivity, and code quality. As these pipelines become more complex, defined by the number of jobs and steps in this paper, there is limited empirical evidence on how this evolution affects key performance metrics. This study addresses this gap by investigating the relationship between pipeline complexity and performance outcomes. By analyzing data from over 194 open-source GitHub repositories, we reveal that while increased complexity generally correlates with longer build durations, the impact on success rates is less direct. More importantly, we found that strategic modifications to pipeline configuration files (i.e., line changes) were frequently associated with significant performance improvements, including shorter build durations and fix times. This research provides guidance for practitioners. Rather than asking whether to increase or decrease pipeline complexity, our findings show that the focus should be on the method of change. We recommend prioritizing small, iterative maintenance activities, which consistently improve performance, over large-scale tool migrations, which have unpredictable outcomes. This approach enables teams to evolve their pipelines while mitigating the risks associated with growing complexity.

1 Introduction

Continuous Integration (CI) has evolved from an emerging practice to a standard approach in modern software development, helping organizations to integrate code changes, run automated tests, and detect integration issues early.

Organizations increasingly adopt complex pipeline configurations for code quality, bug detection, deployment frequency, automated build, and deployment processes, and well-optimized CI pipelines experience significant benefits in deployment reliability, team productivity, and code quality [15].

The landscape of CI/CD is growing increasingly complex. This is evidenced not only by the growing number of different tools and technologies, as highlighted by studies like Giao et al. [4], but also by the internal complexity of individual pipelines. For the purpose of this study, we define structural pipeline complexity by the number of jobs and steps within a single workflow. This trend of growing complexity, both at the ecosystem and pipeline level, reflects organizations' aim to automate more aspects of their software delivery process to achieve rapid and frequent delivery of changes.

The impact of this growing complexity on performance, however, is not always straightforward. For instance, research shows that adding a specific automation tool, such as an automated testing framework, can demonstrably decrease build times [11]. Yet, the net effect of a pipeline's overall structural complexity, encompassing its total number of jobs and steps, on key performance indicators, such as **build duration, build success rate, and fix time**, remains underexplored.

This challenge is compounded in practice by the diverse landscape of CI/CD tools and the complexities of their configuration. Developers must navigate difficult choices between different platforms, each with its own syntax and capabilities. The resulting pipeline configurations, often written in lengthy YAML files, can become difficult to maintain and optimize[5]. Consequently, many teams face a difficult trade-off: adding more jobs and steps can improve test coverage and code quality, but often at the cost of longer feedback cycles and increased maintenance overhead[3]. The lack of clear, empirical data on these trade-offs makes it difficult to make informed decisions[14].

Therefore, to bridge this gap, our research quantitatively investigates how pipeline complexity and its evolution impact build performance. By performing a quantitative analysis over time on historical data from 194 open-source projects, we aim to provide empirical guidance. Specifically, we investigate how different aspects of pipeline configuration, such as the structural complexity (number of jobs and steps) and major evolutionary events (tool migrations and line changes), correlate with key performance indicators: build duration, build success rate, and fix time.

The primary research questions that guide this study are the following:

RQ1: How does CI pipeline complexity correlate with build performance metrics in open-source projects? This primary question aims to establish a foundational understanding of the direct trade-offs associated with pipeline complexity.

RQ2: How do major pipeline evolutionary events impact build performance? Pipelines are not static; they evolve through significant events like tool migrations. As these changes are common in practice [4], this question investigates their direct consequences on performance, aiming to understand the risks and benefits associated with such major evolutionary steps.

RQ3: Which CI/CD activities have the most significant positive impact on build performance? While RQ2 focuses on large-scale changes, pipelines also evolve through small, continuous maintenance activities. This question aims to identify which of these routine activities, such as modifications to configuration files, provide the most significant performance benefits. The goal is to uncover best practices for ongoing pipeline optimization.

To answer these questions, we perform a quantitative analysis on historical data from 194 open-source repositories. Our methodology involves correlation analysis, comparative analysis of performance metrics before and after evolutionary events, and grouping repositories by size (LOC) to provide a detailed view. This study makes the following primary contributions:

1. We provide empirical evidence showing that while increased pipeline complexity primarily lengthens build duration, it does not significantly impact reliability (i.e., success rates and fix times).

2. We demonstrate the contrasting effects of pipeline evolution: large-scale tool migrations yield unpredictable performance outcomes, whereas small, continuous maintenance activities are consistently linked to significant improvements.
3. We provide a data-driven foundation for developers to manage CI/CD pipelines more effectively, emphasizing a strategy of continuous optimization over disruptive, large-scale changes.

The remainder of this paper is organized as follows. Section 2 reviews related work on CI/CD evolution and performance. Section 3 details our research methodology. Sections 4, 5, and 6 present the detailed findings for each research question, respectively. Finally, Section 7 discusses the implications and limitations of our work, and Section 8 concludes the paper.

2 Related Work

This section reviews prior research relevant to our study, focusing on three main areas: the evolution of CI/CD practices, their performance impact, and the resulting research gap. By examining this landscape, we position our research and highlight its unique contribution.

Evolution and Complexity of CI/CD Pipelines Several studies have quantitatively mapped the landscape of CI/CD adoption in open-source projects. For instance, Giao et al. [4] conducted a large-scale analysis revealing a significant increase in CI/CD usage, a dominance of tools like GitHub Actions, and a clear trend of projects migrating between technologies. Similarly, Hilton et al. [9] explored the usage and benefits of continuous integration. Focusing specifically on GitHub Actions, Chen et al. [2] investigated its adoption and common usage patterns, analyzing millions of workflows to identify the most popular triggers and reusable actions. Their work provides a detailed view of how developers construct their workflows on this dominant platform. While these studies provide an excellent overview of which tools are used and that pipelines evolve, they primarily focus on adoption trends rather than the performance consequences of the internal structural changes that accompany this evolution.

Performance Impact of CI Other research has focused on the impact of CI adoption on performance. Studies like Zhao et al. [18] examined the broad effects of CI on development, while Joshi [11] demonstrated that specific automations can significantly decrease build times. Furthermore, Chen et al. explored the impact of adopting GitHub Actions on developer productivity, investigating metrics such as commit frequency and the resolution efficiency of pull requests and issues [2]. These studies confirm that CI practices have a tangible impact. However, by treating CI adoption as a binary choice (yes/no) or focusing on a single, isolated change, this body of work does not address the performance implications of a pipeline's overall structural complexity as it grows over time.

Research Gap The existing literature confirms that CI pipelines are evolving and that CI practices impact performance. However, a clear gap remains at the intersection of these areas. There is limited research that quantitatively links a pipeline's overall structural complexity (e.g., its number of jobs and steps) to a range of performance metrics over time, as existing studies often focus on the general impact of CI adoption rather than its internal structure [18]. Furthermore, how different modes of evolution, such as major tool migrations versus small, continuous changes, distinctly affect performance remains underexplored, even though recent work has confirmed that such evolutionary events are common in practice [4]. Our study addresses this gap by providing a quantitative analysis of the relationship between pipeline structure, evolution, and performance.

3 Methodology

This section outlines the research methodology used to investigate the relationship between CI pipeline complexity and performance. We begin by describing our data collection process, which involved establishing a set of criteria to select a representative sample of 194 open-source repositories from GitHub. Next, we detail how the collected data was structured for analysis, covering our methods for repository grouping, data extraction, and branch classification. Finally, we explain the quantitative analysis methods employed to address research questions. We also formally define the key complexity and performance metrics central to this research, which are summarized in Table 1.

Repository Selection To ensure our dataset was both substantial and representative, we established four primary selection criteria. First, to cover a wide range of development practices, we included projects from diverse and popular programming languages such as Java, Python, and JavaScript, based on usage data from 2024 [6]. Second, to maintain a consistent technical scope, our analysis focused exclusively on projects utilizing GitHub Actions for their CI pipelines. Third, to select for active and collaborative projects where CI provides meaningful value, we required repositories to have at least five contributors, a threshold supported by the findings of Vasilescu et al. [17]. Finally, we set a minimum of 100 stars as a proxy for project maturity and relevance, a heuristic informed by research on repository popularity by Borges et al. [1]. The application of these criteria to repositories active from February 20, 2025, to May 21, 2025, yielded our final dataset of 194 projects.

Data Extraction Our data extraction process began with gathering performance metrics. For each repository, we used the GitHub API to retrieve all GitHub Actions workflow run data, including metadata such as build status and timestamps. This raw data allowed us to compute our primary performance indicators. A formal definition for each metric, including **build duration**, **build success rate**, and **fix time**, is provided in Table 1.

To understand the pipeline's structure and its evolution, we collected data from two additional sources. First, we downloaded all YAML configuration files from the `.github/workflows/` directory to analyze structural complexity, such as the number of jobs and steps. Second, to enable our event-based analyses, we cloned each repository and parsed its full commit history using the `git log` command. This process allowed us to identify all commits that modified CI configuration files and extract key metadata—including

Category	Metric / Term	Definition / Rationale
Components	Jobs	Independent execution units within a CI workflow.
	Steps	Individual commands or actions within a job.
	Tools	The specific CI/CD platforms analysed (e.g., GitHub Actions, Jenkins).
Complexity	Number of Jobs	The total number of jobs defined in a workflow configuration.
	Number of Steps	The total number of sequential steps across all jobs in a workflow.
Performance	Build Duration	The time from a workflow's trigger to its completion.
		Measured as `completed_at - started_at` [8].
	Build Success Rate	The percentage of successful runs out of all completed runs.
		Indicates code stability [16]. Calculated as `(successes / (successes + failures)) * 100`.
	Fix Time	The time between a failed run and the next successful run in the same context.
		Reflecting how quickly developers resolve issues [10]. The context is defined by the same branch, pull-request number, and commit SHA [7].

Table 1: Definitions of Key Metrics and Terminologies

the commit SHA, author, and timestamp—which was crucial for accurately detecting and timing the evolutionary events studied in this research.

Branch Classification To account for differing development workflows, we classified all pipeline runs based on their branch type. Runs occurring on default branches, typically named `main` or `master` as identified from repository metadata, were aggregated and analyzed separately. All other non-default branches, such as feature and development branches, were grouped into a second category. This classification allowed us to compare the pipeline characteristics and performance between stable, primary branches and more active or experimental branches.

LOC-based Grouping To control for project size as a potential confounding variable, we segmented repositories based on their total lines of code (LOC). This approach allows for a more detailed analysis of the relationship between complexity and performance across different project scales. While some research [13] uses fixed LOC values (e.g., 5,000 and 150,000 LOC), we found this approach unsuitable for the skewed distribution of our dataset. Instead, we divided the repositories into four quartile-based groups (Q1-Q4) based on the 25th, 50th, and 75th percentiles of their LOC distribution. This data-driven method created a clear separation in project scale, with the mean LOC for the four groups being approximately 17,600 (Q1), 67,800 (Q2), 223,700 (Q3), and 1,062,000 (Q4), respectively. For each of these groups, we then calculated the key performance metrics separately for default and non-default branches.

Responsible Research In conducting this research, we adhered to ethical guidelines for analyzing public data. All data was collected from publicly accessible open-source repositories on GitHub, and no private or sensitive information was used. We ensured that our analysis and reporting of results were done in a way that respects the work of the open-source contributors and avoids misrepresentation.

4 RQ1: How does CI/CD pipeline complexity correlate with build performance metrics in open-source projects?

To analyze our data, we employed specific statistical methods chosen for their suitability to our research questions and data characteristics. To assess the monotonic relationship between pipeline complexity and performance metrics, we used the Spearman Correlation Coefficient (r). This non-parametric test was selected because our data does not necessarily follow a linear relationship or normal distribution, making Spearman's rank-based correlation the appropriate choice. The associated p-value was used to determine if the observed correlations were statistically significant ($p < 0.05$). Furthermore, to compare the average performance between two distinct groups, specifically, repositories with 'low' versus 'high' pipeline complexity, we conducted an independent samples T-test. This test allowed us to evaluate whether the difference in means for metrics like build duration between these two groups was statistically significant.

Analysis and Findings To address RQ1, we performed a repository-level analysis, aggregating CI/CD pipeline complexity and build performance metrics for each repository. As shown in Table 2, our findings indicate that pipeline complexity is moderately positively correlated with build duration ($r = 0.37$ and $r = 0.40$, $p \ll 0.001$), suggesting that more complex pipelines tend to require longer build times. The correlation with fix time and build success rate, however, was found to be weak and not statistically significant. Notably, the default branch generally demonstrates better performance across all metrics compared to other branches, which suggests that CI/CD optimizations are more heavily focused on the main development line.

To further understand the impact of repository size, we also

Metric Pair	N	Default Mean ± SD	Other Mean ± SD	Spearman r	p-value	T-test p
number of jobs vs build duration	194	27.54 ± 219.53	117.18 ± 1062.76	**0.37**	$3.25 \times 10^{-14***}$	0.8550
number of steps vs build duration	194	27.54 ± 219.53	117.18 ± 1062.76	**0.40**	$1.15 \times 10^{-16***}$	0.1191
number of jobs vs fix time	70	1070.07 ± 3204.50	2712.33 ± 8873.76	-0.26	0.0822	0.0919
number of steps vs fix time	70	1070.07 ± 3204.50	2712.33 ± 8873.76	-0.16	0.299	0.0907
number of jobs vs success rate	194	0.82 ± 0.24	0.75 ± 0.24	-0.10	0.0587	0.5687
number of steps vs success rate	194	0.82 ± 0.24	0.75 ± 0.24	-0.04	0.466	0.2819

Table 2: **Correlation and group comparison of pipeline complexity and build performance metrics**
Note: *** $p < 0.001$

LOC Group	n	Jobs (mean ± SD)	Steps (mean ± SD)	Duration (mean ± SD)	Fix Time (mean ± SD)	Success (mean ± SD)
Q1 (smallest)	48	2.81 ± 3.56	24.40 ± 40.64	2.11 ± 2.93	20.78 ± 239.48	0.92 ± 0.18
Q2	48	2.08 ± 1.32	18.88 ± 14.53	78.42 ± 466.79	226.22 ± 478.22	0.93 ± 0.12
Q3	49	3.14 ± 3.02	32.48 ± 42.50	35.48 ± 158.61	1902.01 ± 5147.68	0.85 ± 0.23
Q4 (largest)	49	3.95 ± 8.32	36.13 ± 110.93	8.96 ± 11.42	124.41 ± 373.81	0.63 ± 0.32

Table 3: **Build performance metrics by LOC quartile group** (default branch)

divided the data by lines of code (LOC) quartile groups. As detailed in Table 3, a clear trend emerged: smaller repositories (Q1) have the shortest build durations and highest success rates, while larger repositories (Q4) show longer build durations and lower success rates. For instance, the smallest quartile (Q1) showed an average success rate of 92%, whereas the largest quartile (Q4) had a rate of 63%. This suggests that repository size is an important factor influencing pipeline performance.

In summary, our analysis for RQ1 reveals two key insights. First, while pipeline complexity is a clear driver of longer build times, its impact on build reliability is surprisingly weak. This suggests that complexity's primary cost is computational rather than a direct introduction of instability. Second, repository size and branch type are significant contextual factors, with smaller projects and default branches consistently exhibiting better performance.

5 RQ2: How do significant pipeline evolutionary events, such as CI tool change, impact performance metrics?

While RQ1 established a baseline correlation, a deeper understanding requires analyzing how pipelines change over time. The evolution of a CI pipeline is not always gradual; it is often marked by significant evolutionary events. In this study, we define these events as CI tool migrations: major, discrete changes where a project switches its primary CI/CD service (e.g., moving from Travis CI to GitHub Actions).

To investigate the impact of these migrations, we first needed to detect them. We identified migration events by analyzing the full commit history of each repository, parsing the `git log` to pinpoint commits where one type of CI configuration file (e.g., `.travis.yml`) was removed and another (e.g., a file in `.github/workflows/`) was added. Once a migration date was identified, we conducted a comparative analysis of performance metrics, collecting data for a period of one month before and one month after the event.

Analysis and Findings The results of our comparative analysis, summarized in Table 4, indicate that CI tool migrations have complex and varied effects on build performance, with no single metric showing universal improvement.

On average, projects experienced a 31.1% increase in build duration post-migration, suggesting a potential learning curve or initial setup overhead associated with new tools. In contrast, the average build success rate generally improved by 18.71%, indicating that migrations may coincide with an effort to improve overall pipeline reliability. However, this trend was not consistent across all projects. The average fix time showed extremely high variance, making it difficult to draw a general conclusion for this metric.

Overall, these findings highlight that CI tool migration is not a guaranteed path to better performance. The outcomes are highly dependent on the specific project context, underscoring the importance of careful, project-specific evaluation before undertaking such a significant change.

6 RQ3: Which CI/CD activities contribute most significantly to pipeline complexity in open-source projects?

To examine the impact of CI/CD activities, specifically, line change events in pipeline configuration files, aligned with their build performance, we conducted a comparative analysis of key performance metrics before and after 1 month for such events. For each repository, we collected build duration, build success rate, and fix time data for the periods immediately preceding and following average line change events, following our methodology. For more detailed tables, visit the GitHub repository [12].

	Before (s)	After (s)	Change Rate
BuildDuration	0.68	0.32	-52.9%
FixTime	202.61	121.41	-40.1%
SuccessRate(%)	82.18	83.89	2.1%

Table 5: Average build performance metrics before and after line change events.

Table 4: Performance metrics before and after CI tool migration events.

Repo & Date		Before Migration			After Migration			Change (%)		
repo	date	duration	fix_time	success_rate	duration	fix_time	success_rate	duration	fix_time	success_rate
AlexProg_rammerDE/SoulFire	2025-04-01	0.03	15.0	98.9	0.03	416.6	85.0	0.0	2677.33	-14.02
deepspeedai/DeepSpeed	2025-02-24	2.0	1.4	85.2	1.6	5.5	90.0	-20.0	292.86	5.63
hiero-ledger/hiero-sdk-java	2025-01-29	1.8	771.0	63.7	4.4	20.5	72.0	144.44	-20.54	13.05
pandas-dev/pandas	2025-01-21	0.2	0.0	55.3	0.2	151.6	94.1	0.0	0.0	70.16
Mean	Mean	1.01	196.85	75.78	1.56	148.05	85.28	31.11	996.91	18.71

Analysis and Findings Our analysis of ongoing maintenance focused on identifying projects with significant line changes in their CI/CD configuration files. Interestingly, out of our entire dataset, we found 17 projects that met the criteria for such a notable change event. This amount itself is a noteworthy finding, suggesting that CI configurations in many projects may remain relatively static after initial setup, or that changes are typically minor and incremental rather than substantial.

For this subset of projects, the performance impact of these changes was both positive, as summarized in Table 5. We observed a remarkable 52.9% average reduction in build duration and a 40.1% reduction in fix time. This stands in contrast to the unpredictable outcomes of the large-scale tool migrations analyzed in RQ2. The consistent, positive impact of these smaller, targeted modifications suggests that continuous maintenance is a highly effective and low-risk strategy for improving pipeline performance. The modest 2.1% increase in success rate further supports the conclusion that these changes enhance not only efficiency but also reliability.

7 Discussion

This section interprets the findings from our quantitative analysis, discusses their implications, and acknowledges the study's limitations and future research directions.

Interpretation of Key Findings Our quantitative analysis yielded several key findings that provide a nuanced view of CI/CD practices. The analysis for RQ1 demonstrated that while increased pipeline complexity has a strong, statistically significant correlation with longer build durations (e.g., $r = 0.40$ for steps), its link to reliability metrics like success rate and fix time is weak. This finding suggests that the primary cost of complexity is computational, rather than an inherent introduction of instability, perhaps because added steps are often accompanied by more robust validation practices.

Furthermore, our results highlight a stark contrast between two modes of pipeline evolution. For RQ2, we found that major tool migrations led to unpredictable outcomes, including an average 31.1% increase in build duration despite an average improvement in success rate. In contrast, for RQ3, we found that small, ongoing maintenance activities were linked to dramatic improvements, such as an average build duration reduction of 52.9%. This evidence strongly suggests that how a pipeline evolves—through iterative refinement versus large-scale migrations—is a more critical performance factor than the tools themselves.

Implications of the Study For practitioners, our findings offer clear, actionable guidance. The results strongly suggest that substantial performance gains are more reliably achieved through continuous, targeted optimization of existing pipelines rather than through large-scale, disruptive tool migrations. Teams should carefully evaluate whether the expected benefits of a new tool outweigh the risk of disruption and longer build times. The emphasis for efficient CI/CD should be on strategic, ongoing maintenance.

For researchers, this study underscores the importance of analyzing the internal structure and evolution of pipelines, moving beyond binary metrics of CI adoption. Our findings open new questions, particularly regarding the causal mechanisms that lead to varied outcomes in tool migrations and the specific types of configuration changes that yield the most significant benefits.

8 Threats to Validity

While our study provides valuable insights, it is important to acknowledge its limitations, which we frame as threats to validity.

External Validity The generalizability of our findings is constrained by our data source. Our analysis is based on public, open-source repositories using GitHub Actions. Therefore, the observed patterns may not fully represent practices in private industrial settings or on other CI/CD platforms such as GitLab CI or Jenkins.

Internal Validity Our quantitative approach identifies strong correlations but cannot establish causality. For instance, while we observed that line changes in configuration files correlate with performance improvements, we cannot definitively conclude that these changes caused the improvements. Other concurrent factors, such as library updates or infrastructure changes, could have contributed to the observed effects.

Construct Validity Finally, we faced challenges in accurately measuring our core constructs. Our build duration calculation can be skewed by queuing times, meaning it may not always reflect the true execution time. Similarly, the presence of dedicated test branches with an inherently high failure rate could artificially lower the average build success rate for some repositories, potentially misrepresenting their overall stability.

9 Conclusion and Future Work

This research set out to understand the evolution of CI pipeline complexity and its impact on build performance. We addressed three primary questions: (1) how pipeline complexity correlates with performance, (2) the impact of major evolutionary

events like tool migrations, and (3) the effect of smaller, ongoing maintenance activities.

Our analysis provided several key conclusions. We demonstrated that while structural complexity (i.e., the number of jobs and steps) is a clear driver of longer build times, it has a weak link to build reliability. The study's main contribution is the finding that the method of evolution is a more critical performance factor than complexity itself. We found that large-scale tool migrations have varied and unpredictable outcomes, whereas continuous, iterative maintenance consistently yields significant performance benefits. This provides empirical evidence that effective pipeline management relies on a strategy of ongoing optimization over disruptive changes.

Based on these findings and the limitations of our study, several avenues for future research emerge. A natural next step is to expand this analysis to different contexts, such as private industry projects and other CI/CD platforms, to improve generalizability. Incorporating qualitative methods, like developer surveys, could provide deeper context on the rationale behind pipeline evolution. Furthermore, developing more robust performance metrics, particularly for build duration, would enhance the precision of future studies in this area.

References

[1] Hudson Borges, Andre Hora, and Marco Tulio Valente. On the popularity of github applications: A preliminary note. *arXiv preprint arXiv:1507.00604*, 2015.

[2] Tingting Chen, Yang Zhang, Shu Chen, Tao Wang, and Yiwen Wu. Let's supercharge the workflows: An empirical study of github actions. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 01–10, 2021.

[3] CircleCI. How circleci uses an internal developer portal built with backstage, 2023. Accessed: 2025-06-17.

[4] Hugo da Gião, André Flores, Rui Pereira, and Jácome Cunha. Chronicles of ci/cd: A deep dive into its usage over time, 2024.

[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, July 2022.

[6] GitHub. The state of the octoverse 2024. In *GitHub News & Insights*, 2024. Accessed: May 22, 2025.

[7] GitHub. Github glossary. In *GitHub Documentation*, 2025. Accessed: May 31, 2025.

[8] GitHub. Workflow runs - github docs. In *GitHub REST API Documentation*, 2025. Accessed: May 31, 2025.

[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. ACM, 2016.

[10] JetBrains. Measure ci/cd performance with devops metrics. Accessed: 2025-04-22.

[11] Nikhil Yogesh Joshi. Implementing automated testing frameworks in ci/cd pipelines: Improving code quality and reducing time to market. *International Journal on Recent and Innovation Trends in Computing and Communication*, 10(6):106–113, 2022.

[12] lkdmc (Kwangjin Lee). Rp-python-ver. GitHub repository, 2025. Accessed: 2025-06-13.

[13] Md A A Mamun, Christian Berger, and Jörgen Hansson. Effects of measurements on correlations of software code metrics. *Empirical Software Engineering*, 24(5):2764–2818, 2019.

[14] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

[15] Kartheek Medhavi Penagamuri Shriram. Engineering efficiency through ci/cd pipeline optimization. 2025.

[16] Software.com. Test pass rate, n.d. Accessed: 2025-04-25.

[17] Bogdan Vasilescu, Stef van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark G.J. van den Brand. Continuous integration in a social-coding world: Empirical evidence from github. In *2014 IEEE International Conference on Software Maintenance and Evolution*, page 401–405. IEEE, September 2014.

[18] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: A large-scale empirical study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 60–71. IEEE, October 2017.