PrioTestCI: Efficient Test Case Prioritization in GitHub Workflows for CI Optimization

Shubham Vasudeo Desai*, Shonil Bhide*, Souhaila Serbout†, Luciano Marchezan‡, Wesley K. G. Assunção*
*North Carolina State University, Raleigh, United States
†University of Zurich, Zürich, Switzerland
‡University of Montreal, Montreal, Canada

***Abstract*—Continuous Integration (CI) is a widely adopted practice in software development to automatically verify code changes across diverse environments. However, executing the full test suite on every pull request update can lead to redundant runs, slower feedback loops, and inefficient utilization of CI resources. To address this issue, we introduce PrioTestCI, a prioritization technique within GitHub Actions that focuses on re-executing test cases that have previously failed. If these prioritized tests succeed, the remaining tests proceed; otherwise, the workflow terminates early, saving computation resources and providing early feedback to developers. PrioTestCI utilizes commit-to-commit test result tracking to inform future test runs, thereby reducing unnecessary repetition and accelerating validation cycles. We evaluated our technique on the Pytest project, a real-world open-source project with an extensive test matrix. PrioTestCI resulted in a CI runtime reduction of 1h57m39s compared to the normal workflow, with individual configuration improvements ranging from 63.75% to 91.94% (81.55% on average). Demo video: https://youtu.be/_3CF9LJdv0I?si=XyE_8mBnDxk1lMnD Repository: https://github.com/ShubhamDesai/CI-Optimization**

I. Introduction

In modern software development, Continuous Integration (CI) plays a central role in maintaining code quality by automatically validating changes across diverse environments [9]. CI facilitates rapid feedback, enabling developers to detect and resolve bugs and integration issues early [5, 14]. The adoption of CI has been accelerated by the GitHub Actions platform,[1] which offers seamless integration within the GitHub ecosystem, making CI widely used by developers [13]. For instance, the GitHub CI solution had an adoption rate of 44% in 2022 [4], surpassing popular external CI services such as Travis CI,[2] only 18 months after its launch [9, 12].

While existing CI platforms have lowered the bar to CI adoption, they have limitations in scaling with increasing resource requirements [15], with still many opportunities for improvement [1]. For instance, GitHub Actions comes with resource and cost-related constraints, which can limit its scalability for large-scale or highly active repositories [8]. Free-tier users are restricted to a maximum of 20 concurrent jobs, with an even lower cap of five for macOS jobs, and each job has a maximum runtime limit of six hours [7]. Furthermore, GitHub enforces strict queuing rules, allowing no more than 500 workflow runs per repository within any 10-second interval [8]. While public repositories have unlimited CI usage, private repositories are subject to billing once the monthly quota—typically 2,000 minutes for free accounts—is exceeded. The cost structure varies by operating system: Linux jobs are charged at $0.008 per minute, Windows jobs cost twice as much at $0.016, and macOS jobs are the most expensive at $0.08 per minute [6].

To make matters worse, as projects grow in complexity, the cost and the time required to run full test suites on every code change increase substantially [10]. However, not every change affects all parts of the codebase equally. Despite that, traditional CI workflows often treat test cases uniformly and re-trigger all of them by default on each triggering event (e.g., push, pull_request). These limitations underscore the importance of intelligent test strategies in CI workflows.

Due to the importance of CI, studies have investigated this topic. Cho et al. [3] introduced AFSAC, a history-based test prioritization method that leverages prior failure data. Their method ranks test cases based on past failure frequency (static phase) and refines the order using failure correlations (dynamic phase). AFSAC was evaluated on real-world projects, such as Apache Tomcat and Camel, to improve fault detection. Khan et al. [11] proposed a framework that automates feature extraction, model selection, and hyperparameter tuning for training machine learning models to automatically prioritize test cases for CI. Their goal is to perform early fault detection and prioritize the test cases accordingly. Furthermore, recent studies have shown that optimizing CI/CD with platforms such as GitHub can bring benefits; however, these benefits are currently limited due to factors such as the lack of standardization, the need for better automation, and inadequate tooling support [1, 2].

The aforementioned studies reinforce the need for better strategies to maximize the use of CI workflows such as GitHub Actions. In this paper, we propose a history-based test case prioritization technique (PrioTestCI), which focuses resources on the most failure-prone tests early in the workflow. PrioTestCI minimizes execution time and delays while improving the overall cost efficiency of CI workflows. We evaluated PrioTestCI on the Pytest[3] project, leading to a CI total runtime reduction of *1h57m39s* in comparison to the normal workflow, ranging from 63.75% to 91.94% considering individual configurations (81.55% on average).

[1]https://github.com/solutions/use-case/ci-cd
[2]https://www.travis-ci.com/
[3]https://github.com/pytest-dev/pytest/

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00365

II. Motivating Example

Active software projects, such as open-source ones, often receive many pull requests (PRs) daily. To be ready to merge, each PR is required to pass test cases in the CI workflow. Additionally, ensuring cross-platform compatibility is essential, but contributors typically cannot test their changes across all operating systems (e.g., Windows, Linux, and macOS). A PR that works locally (in Windows) may fail on another platform (e.g., macOS or Linux). Thus, issues (i.e., bugs) only surface during CI, and resolving them often involves multiple trial-and-error commits, resulting in repeated and lengthy CI workflow runs. With large test suites, multiple platforms, and GitHub's workflow concurrency limits, this can lead to long execution times and PR queuing.[4]

In the best case, an experienced developer resolves the issue in a single follow-up commit, requiring two full workflow runs. However, less experienced contributors push several commits to diagnose and fix the issue. If the failure happens late in the workflow (e.g., after running the entire test suite before detecting the error), this wastes the developer's time and computing resources. Meanwhile, other PRs stay queued due to limited workflow concurrency. These inefficiencies are evident in real-world scenarios:

- **OSGeo/GRASS GIS (PR #4292):**[5] In this PR, the contributor pushed multiple commits (`87bd0fa`, `645fd13`, `3261cb0`, `5f43116`) to address the same issue. Each full workflow run took approximately two hours, consuming significant compute resources and delaying other PRs that were queued.
- **Pytest Repository (PR #13380):**[6] In this case, the first commit (`fa05162`) triggered a full workflow. The second commit (`1e40e83`), which only updated the author's name and did not modify the codebase, still initiated a new workflow run. In this subsequent run, instead of failing early, the workflow proceeded to execute several test cases before ultimately encountering the same failure. This unnecessary execution could have been avoided through proper test case prioritization.

These examples reveal inefficiencies in traditional CI workflows, where repeated failures still trigger full test suite runs. While some open-source projects terminate workflows after a failure threshold, redundant executions still occur. This highlights the need for history-based test prioritization to save time, reduce compute usage, and improve developer experience.

III. Proposed Solution

To address the inefficiencies of redundant test execution in CI workflows, we present PrioTestCI, a history-based test case prioritization technique integrated into GitHub Actions. We assume all test cases are independent of each other. PrioTestCI prioritizes test cases that failed in previous workflow runs, leading workflows to fail fast on known issues. This saves time and resources. The novelty of our solution lies in its platform- and language-independence, enabling it to be adopted in any project that utilizes GitHub Actions for testing, regardless of the programming language or testing framework employed. Moreover, the underlying logic of prioritizing based on historical failures is generalizable and can be applied beyond GitHub Actions in other CI environments.

[4]https://github.com/orgs/community/discussions/147604
[5]https://github.com/OSGeo/grass/pull/4292/commits
[6]https://github.com/pytest-dev/pytest/pull/13380/commits

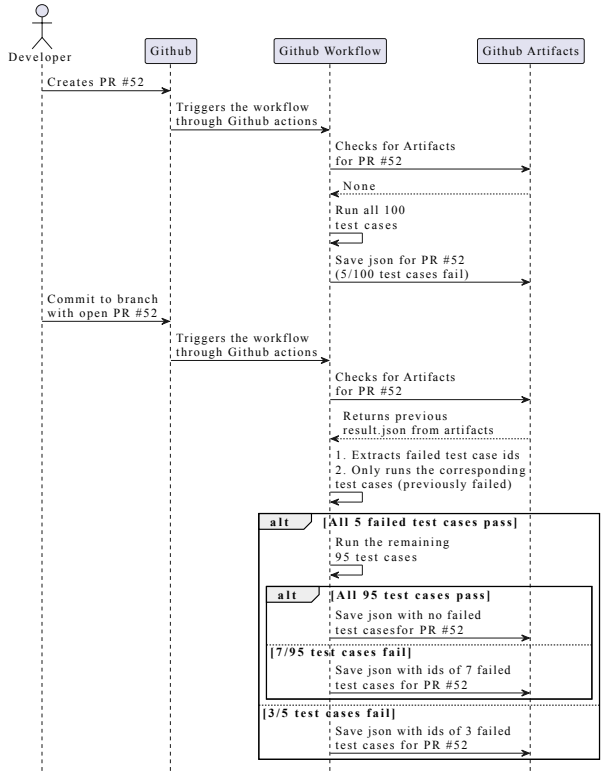

Fig. 1. UML sequence diagram illustrating the proposed workflow logic with GitHub Actions and artifact-based test prioritization.

A. System Overview

An overview of the logic of PrioTestCI is presented in Figure 1, operating in six steps: (i) When a PR is created or updated, the CI workflow is automatically triggered. (ii) On the initial run, the technique executes the full test suite (e.g., one hundred tests). Any failing test cases (e.g., five tests) are recorded using their unique node identifiers and stored as a GitHub Artifact in a persistent JSON format. (iii) For subsequent commits on the same PR, PrioTestCI fetches the created artifact and identifies previously failed test cases. (iv) These (five) failed tests are prioritized and executed first. If they fail again, the workflow terminates early, saving the overhead of executing additional passing test cases (the other 95). (v) If all prioritized tests pass, the remaining tests are executed to ensure overall correctness. Finally, (vi) the artifact is updated after every run to reflect the latest test outcomes, maintaining a persistent, adaptive failure history.

B. Architecture and Implementation

PrioTestCI's architecture comprises four components, designed to be modular and easily adaptable across repositories:

GitHub Actions Orchestrator: Defines the core workflow logic using YAML, enabling seamless integration into any GitHub-based repository regardless of the codebase language. It triggers workflows on PR events and handles matrix job execution across different platforms (Windows, Linux, macOS).

Artifact Storage: Leverages GitHub's built-in artifact system to store JSON files that contain failed test identifiers. These artifacts persist for up to 90 days (due to GitHub Action rules) and serve as the memory for history-based prioritization.

Artifact Scoping and Correctness: To maximize accuracy and avoid cross-contamination, artifact storage is scoped by both PR ID and matrix job configuration. Artifacts are saved as `artifacts/pr-${PR_ID}/${WORKFLOW_ID}/`, allowing precise retrieval per job (e.g., `ubuntu-py312`). This separation ensures correctness and prevents result leakage across jobs or commits.

Prioritization Engine: Encoded directly within the GitHub Actions YAML workflow, the prioritization logic checks for artifacts from previous runs. In case failure artifacts are found, PrioTestCI first re-executes only the previously failed test cases. If those pass, the remaining tests in the suite are executed. If no prior failures exist, the full test suite is run. This conditional execution enables early failure detection, conserves resources, and provides developers with early feedback.

C. Batching

Executing large test suites via GitHub Actions can exceed OS-specific command-line length limits, especially when referencing thousands of test cases. Errors like `Argument list too long` may occur, depending on the platform. To prevent such failures, PrioTestCI splits the test suite into smaller batches (typically 20–50 test cases per batch, which can be configured). Python scripts automate this process by generating batch files, which are then executed sequentially. PrioTestCI's batching approach ensures that: (i) command length limits are never exceeded; (ii) execution order is preserved within and across batches; (iii) logs remain clean and easy to debug; and (iv) failed or flaky tests can be isolated and re-executed.

D. Summary

PrioTestCI offers a lightweight, language-agnostic solution to reduce CI runtime by intelligently reordering test execution based on prior CI test case failures. Its modular architecture (built around GitHub Actions workflows, scoped artifact storage, and dynamic batching) ensures compatibility with real-world, matrix-based repositories while maintaining correctness and scalability.

IV. Evaluation

To assess the efficiency of our technique, we applied it to the Pytest project, which has a complex GitHub Actions matrix with multiple operating systems (Windows, Ubuntu, macOS) and Python versions. We replicated this setup while excluding the `macos-py39` configuration, resulting in 19 test-executing jobs. Workflows without test cases were also excluded.

TABLE I
Execution Time Comparison: Original vs. Forked Repo

Configuration	Original	Forked (Ours)	Time Saved (%)
windows-py39-unittestextras	10m 24s	1m 50s	8m 34s (82.37%)
windows-py39-pluggy	7m 4s	1m 36s	5m 28s (77.35%)
windows-py39-xdist	6m 40s	2m 25s	4m 15s (63.75%)
windows-py310	7m 32s	2m 25s	5m 7s (67.92%)
windows-py311	10m 23s	2m 27s	7m 56s (76.4%)
windows-py312	10m 25s	1m 53s	8m 32s (81.92%)
windows-py313	10m 23s	1m 38s	8m 45s (84.27%)
ubuntu-py39-lsof-numpy-pexpect	10m 32s	1m 4s	9m 28s (89.87%)
ubuntu-py39-pluggy	5m 41s	59s	4m 42s (82.69%)
ubuntu-py39-freeze	46s	48s	-2s (–)
ubuntu-py39-xdist	5m 27s	48s	4m 39s (85.32%)
ubuntu-py310-xdist	5m 50s	50s	5m 0s (85.71%)
ubuntu-py311	10m 33s	51s	9m 42s (91.94%)
ubuntu-py312	9m 44s	1m 4s	8m 40s (89.04%)
ubuntu-py313-pexpect	10m 31s	1m 2s	9m 29s (90.17%)
ubuntu-pypy3-xdist	8m 15s	1m 20s	6m 55s (83.83%)
macos-py310	5m 27s	1m 3s	4m 24s (80.73%)
macos-py312	3m 47s	52s	2m 55s (77.09%)
macos-py313	4m 5s	55s	3m 10s (77.55%)
Total	2h 23m 29s	25m 50s	1h 57m 39s (avg. 81.55%)
Min Time Saved (%)	—	—	4m 15s (63.75%)
Max Time Saved (%)	—	—	9m 42s (91.94%)

Our evaluation focused on PR #13380 (discussed in Section II) from the original Pytest repository, specifically the first two commits: (i) **Commit 1 (fa05162)**, which introduced functional code changes; and (ii) **Commit 2 (1e40e83)**, which have a non-functional update (author name addition), but yet encountered test failures. Since the second commit did not modify the codebase, repeating the full test suite was unnecessary. However, in the original CI workflow, both commits triggered complete test executions, resulting in redundant computation and delayed feedback, which highlights the value of our test prioritization approach.

To simulate this PR in our fork and evaluate PrioTestCI's behavior, we followed a controlled branching strategy:

- A new branch was created from the latest commit of PR #13380 to replicate the state of the PR.
- To test our technique against both commits, we created a separate branch with the first commit as its HEAD. A PR was opened from this branch.
- On the first run, the full test suite was executed. Failed tests were recorded and stored as artifacts.
- On the second commit, our tool retrieved previously failed test cases and ran only those first. Since the failures persisted, the workflow stopped early, demonstrating PrioTestCI's efficiency.

The results in Table I demonstrate a consistent and substantial reduction in CI execution time across all configurations, except for `ubuntu-py39-freeze`, having a negligible increase of 2 seconds. The most significant improvement was observed in the `ubuntu-py311` configuration, achieving a 91.94% reduction in runtime. The least significant time saving (excluding the outlier) was for the `windows-py39-xdist` job, with a reduction of 63.75%. Overall, the total CI execution time dropped from *2h23m29s* to *25m50s*, resulting in a cumu-

lative time saving of *1h57m39s*, with an average improvement of 81.55% per configuration. These findings confirm that our history-based test case prioritization approach significantly improves CI efficiency.

V. Discussions

A. Limitations and Future Enhancements

While PrioTestCI demonstrates significant improvements in CI efficiency, a few limitations present opportunities for future enhancement:

Repository Scope: The current evaluation focuses on the Pytest project, which is representative of large, matrix-heavy open-source projects. Future work could extend the application to additional repositories and testing frameworks to further validate the tool's adaptability.

Manual Integration: Incorporating the tool into existing CI workflows currently requires manual editing of GitHub Actions workflow files. Developing an automated configuration utility would improve usability and facilitate broader adoption across projects.

Artifact Retention Limit: GitHub Actions retains artifacts for a maximum of 90 days. Beyond this period, any stored artifacts, including test results, are automatically deleted. As a result, older PRs may no longer have access to previously failed test information, limiting the effectiveness of prioritization for long-lived or inactive PRs. This is also a limitation in our evaluation.

B. Future Work

We plan to evaluate our approach on a broader set of open-source projects to assess generalizability regarding CI configuration and the number of test cases. We also plan to investigate how to automate the integration process with other repositories. This would reduce the manual setup required to run PrioTestCI. While test prioritization currently operates at the PR level, we aim to incorporate cross-PR failure history for repository-level trend detection. Thus, our approach can utilize the history from other PRs that may contain the same set (or subset) of test cases. Additionally, we plan to address the 90-day artifact persistence limit by persisting artifacts outside of GitHub Actions, only accessing them during the CI workflow. Lastly, we plan to improve the reporting results by considering various characteristics, such as version mismatches or environmental drift.

VI. Concluding Remarks

PrioTestCI significantly reduces CI runtime by enabling early termination of workflows based on historical test failures. This not only accelerates feedback for developers but also frees up CI resources for more critical PRs, improving overall system efficiency. Additionally, given that GitHub-hosted runners are backed by physical infrastructure, optimizing workflow executions contributes to better utilization of server resources. The results of our evaluation show that PrioTestCI can be used to optimize these resources, since it was able to reduce the runtime of a Pytest PR by 81.55% on average. In turn, this promotes energy efficiency and supports sustainable computing practices, which is an increasingly important consideration in large-scale software development.

VII. Data Availability

Source code and results are available at: https://github.com/ShubhamDesai/CI-Optimization. For external links and detailed references, see the repository's `README.md`.

References

[1] Islem Bouzenia et al. "Resource usage and optimization opportunities in workflows of github actions". In: *ICSE*. 2024, pp. 1–12.

[2] Guillaume Cardoen et al. "A dataset of GitHub Actions workflow histories". In: *21st MSR*. 2024, pp. 677–681. DOI: 10.1145/3643991.3644867.

[3] Younghwan Cho et al. "History-Based Test Case Prioritization for Failure Information". In: *23rd APSEC*. IEEE, 2016, pp. 385–388.

[4] Alexandre Decan et al. "On the use of github actions in software development repositories". In: *ICSME*. IEEE. 2022, pp. 235–245.

[5] João Faustino et al. "DevOps benefits: A systematic literature review". In: *Software: Practice and Experience* 52.9 (2022), pp. 1905–1926.

[6] GitHub. *About billing for GitHub Actions*. https://docs.github.com/en/billing/managing-billing-for-github-actions/about-billing-for-github-actions. 2025.

[7] GitHub. *Usage limits, billing, and administration*. https://docs.github.com/en/actions/learn-github-actions/usage-limits-billing-and-administration. 2025.

[8] *GitHub Actions limits*. https://docs.github.com/en/actions/reference/actions-limits. 2025.

[9] Mehdi Golzadeh et al. "On the rise and fall of CI services in GitHub". In: *SANER*. IEEE. 2022, pp. 662–672.

[10] Xianhao Jin et al. "A cost-efficient approach to building in continuous integration". In: *ICSE*. 2020, pp. 13–25.

[11] Md Asif Khan et al. "ML-Based Test Case Prioritization: A Research and Production Perspective in CI Environments". In: *IEEE ICST*. 2025, pp. 476–486.

[12] Pooya Rostami Mazrae et al. "On the usage, co-usage and migration of CI/CD tools: A qualitative analysis". In: *Empirical Software Engineering* 28.2 (2023), p. 52.

[13] Sk Golam Saroar et al. "Developers' perception of github actions: A survey analysis". In: *27th EASE*. 2023, pp. 121–130.

[14] Fiorella Zampetti et al. "An empirical characterization of bad practices in continuous integration". In: *Empirical Software Engineering* 25.2 (2020), pp. 1095–1135.

[15] Lianyu Zheng et al. "Why Do GitHub Actions Workflows Fail? An Empirical Study". In: *ACM Transactions on Software Engineering and Methodology* (2025).