Resource Usage and Optimization Opportunities in Workflows of GitHub Actions

Islem Bouzenia
University of Stuttgart
Stuttgart, Germany
fi_bouzenia@esi.dz

Michael Pradel
University of Stuttgart
Stuttgart, Germany
michael@binaervarianz.de

ABSTRACT

Continuous integration and continuous delivery (CI/CD) has become a prevalent practice in software development. GitHub Actions is emerging as a popular platform for implementing CI/CD pipelines, called workflows, especially because the platform offers 2,000 minutes of computation for free to public repositories each month. To understand what these resources are used for and whether CI/CD could be more efficient, this paper presents the first comprehensive empirical study of resource usage and optimization opportunities of GitHub Action workflows. Our findings show that CI/CD imposes significant costs, e.g., $504 per year for an average paid-tier repository. The majority of the used resources is consumed by testing and building (91.2%), which is triggered by pull requests (50.7%), pushes (30.9%), and regularly scheduled workflows (15.5%). While existing optimizations, such as caching (adopted by 32.9% of paid-tier repositories), demonstrate a positive impact, they overall remain underutilized. This result underscores the need for enhanced documentation and tools to guide developers toward more resource-efficient workflows. Moreover, we show that relatively simple changes in the platform, such as deactivating scheduled workflows when repositories are inactive, could result in reductions of execution time between 1.1% and 31.6% over the impacted workflows. Overall, we envision our findings to help improve the resource efficiency of CI/CD pipelines.

CCS CONCEPTS

• **Software and its engineering** → *Development frameworks and environments*; *Software libraries and repositories*.

ACM Reference Format:
Islem Bouzenia and Michael Pradel. 2024. Resource Usage and Optimization Opportunities in Workflows of GitHub Actions. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3623303

1 INTRODUCTION

GitHub Actions is a powerful tool for developers to automate and streamline their continuous integration and continuous delivery (CI/CD) processes. Since its release in 2019, it has been widely adopted by the community of developers, with more than 30% of open-source projects on GitHub utilizing it to automate their CI/CD workflow [9, 23]. GitHub Actions enables developers to create workflows that are triggered by events, such as pushes, pull requests, and the opening of issues. Developers can customize workflows to perform a variety of tasks, e.g., building and testing code, deploying to various environments, and automatically creating releases.

Most workflows execute on virtual machines (VMs) hosted by GitHub, many of which are offered for free to public projects. The computational resources offered to a free account include up to 2,000 CPU minutes (33 hours) per month on virtual machines with 7GB of main memory, 14GB of disk storage, and two or three CPU cores depending on the operating system. We refer to a repository as *free-tier* if it uses only freely available resources, and as *paid-tier* otherwise. Virtual machines also allow developers to test their code in a variety of environments and configurations, which improves the dependability and stability of their software. Thanks to its integration into the GitHub ecosystem and the offered computing resources, GitHub Actions has quickly emerged as a dominant CI/CD platform [15].

This popularity has motivated researchers to study the level of adoption of GitHub Actions, the repositories that use it, and the tasks executed therein [7, 9, 23, 39]. These studies are based on statically analyzing workflow specifications, e.g., to quantify the workflow scripts, jobs, and corresponding tasks. Other research focuses on scrutinizing and characterizing the security of GitHub Actions in comparison to other CI/CD platforms [4, 24].

An important aspect neglected by the existing literature are the computing resources consumed on the platform. In particular, it is currently unknown how many computing resources projects typically consume, what these resources are used for, and what kinds of optimization opportunities exist. Obtaining insights into resource usage and optimization opportunities in workflows on GitHub Actions helps understand their impact on development costs, and ultimately, contributes toward cheaper and faster CI/CD.

This paper presents the first empirical study that analyzes computational resource usage and optimization opportunities of workflows. Our study is based on a dataset of 952 repositories that performed 1.3 million workflow runs over a period of 30 months, which is the largest such dataset we are aware of. The study is driven by three research questions:

RQ1: What amount of resources do workflows use and what are these resources used for? The average annual cost of running workflows for a paid-tier repository in our dataset is approximately $504, with the majority of executed jobs running on the least expensive Linux machines. Pull requests, pushes, and scheduled events are the most common triggering events for workflows, accounting for more than 90.0% of the total VM time. The most frequently performed tasks are testing and building. While most workflows complete successfully, 10.0% to 17.4% fail, primarily due to errors in code during testing or building. These results show the considerable costs incurred by CI/CD platforms, which motivates studying possible optimizations.

RQ2: To what extent do workflows use currently available optimization mechanisms, and what is their impact on the consumed resources? We study and analyze both the prevalence and impact of six mechanisms that could help reduce the cost of running workflows. The mechanisms range from sophisticated optimizations, such as caching, which is used by 32.9% of paid-tier repositories and reduces VM time by 3.4%, to simple configuration options, such as setting a timeout value for a job, which is used by 14.0% of paid-tier repositories, impacts 4.3% of runs, and reduces VM time by 8.1%.

RQ3: What further optimization opportunities exist and how much could they impact resource usage? In addition to the existing optimization mechanisms, we investigate four techniques that currently are not present on GitHub Actions. These optimizations could impact up to 4.5% of all runs and save up to 3.5% of total VM time, complementing the existing options studied in RQ2. These findings show that there still are significant opportunities for reducing both computational and monetary costs.

The results of this study could interest at least two groups of people. On the one hand, it will help developers in reducing the time taken by their CI/CD pipelines, which decreases waiting time and speeds up the development process. In addition, repositories that require resources beyond those provided for free by GitHub have an interest in reducing the monetary costs imposed by workflows. On the other hand, our results can guide platform providers, such as GitHub, when reducing their costs, especially the costs imposed by free-tier repositories.

In summary, this paper presents the following contributions:

(1) The first study of the resource usage of workflows on GitHub Actions, which differs from prior work that studies static properties of workflows [7, 9, 23, 39].
(2) A quantitative analysis of existing and potential optimizations of CI/CD pipelines that shows the (potential) prevalence and impact of ten different optimizations.
(3) Actionable insights on how to reduce the computational resources consumed by workflows, which is relevant for both developers and platform providers.
(4) A reusable dataset of 1.3 million workflow runs from 952 repositories, which can serve as a basis for further analysis and novel optimization techniques.

2 BACKGROUND ON GITHUB ACTIONS

GitHub Actions is a platform for automating CI/CD (continuous integration/continuous deployment) tasks within the GitHub ecosystem. Developers use the platform, e.g., for automatically executing tests on every new commit or for regularly deploying applications. The central concept of the platform are *workflows*, which are executable processes that perform one or more *jobs* when a specific *trigger event* occurs. The jobs within a workflow are either executed in parallel, sequentially as declared in the workflow, or in an order constrained by dependencies between jobs. Each job consists of sequentially executed *steps*, each representing either a command, such as executing a bash script, or an *action*, i.e., reusable code that implements a common CI/CD-related tasks, e.g., checking out code. The triggering events that start workflows include pushing to a repository, creating a new pull requests, and also scheduled events that happen regularly, similar to a cron job.

Jobs are executed on *virtual machines (VMs)*, where each job is executed inside its own VM runner. That is, two jobs cannot be executed in parallel on the same VM. We call each execution of a job a *job run* or simply a *run*. A workflow execution includes at least one job run. GitHub offers VMs with different versions of Linux, macOS, and Windows, and also supports self-hosted VMs on a user's own infrastructure.

All computations performed on GitHub-hosted VMs are billed on a per-minute basis. For example, as of March 2023, running a minute of computation on a standard Linux VM (2 CPUs, 7GB RAM, 14GB disk space) costs $0.008. In addition to CPU time, GitHub also charges costs for disk space,[1] which we do not account for in this study. GitHub offers free resources to public repositories, e.g., 2,000 free minutes of computation to each unpaid user account. To help developers manage the CI/CD costs, GitHub Actions offers several optimization mechanisms, e.g., caching, failing quickly if one out of a set of related jobs fails, or skipping workflows for particular trigger events.

3 METHODOLOGY

To answer the research questions listed in Section 1, we gather a dataset of 1.3 million workflow runs from 952 open-source repositories on GitHub. To the best of our knowledge, this dataset is the largest of its kind. The following describes our methodology for gathering the dataset (Section 3.1), a set of metrics to quantify the resource usage of workflows (Section 3.2), and a name-based analysis to determine what CI/CD tasks a job performs (Section 3.3) followed by the methodology for estimating the prevalence and impact of optimizations (Section 3.4). Figure 1 gives an overview of our methodology.

3.1 Data Collection

We collect a dataset of workflow runs in three steps. First, we sample 600 repositories from a set of 67K repositories collected by prior work [9], which have at least one workflow, 100 stars, and 100 commits. To also study less popular projects, we augment the initial set with 352 further repositories, which we obtain by randomly sampling 74K repositories with less than 100 stars from a larger set of 735K repositories [8] and by then keeping only those that are using workflows. The combined list results in a dataset of 952 repositories. Second, we use the GitHub API to obtain all workflow runs for each selected repository. The gathered information includes the date and time of each workflow run, the workflow ID, the branch or commit that triggered the workflow, and other metadata. Third, we use the GitHub API to gather details about the jobs that were executed within each workflow run. These data include, e.g., the

[1]https://docs.github.com/en/billing/managing-billing-for-github-actions/about-billing-for-github-actions

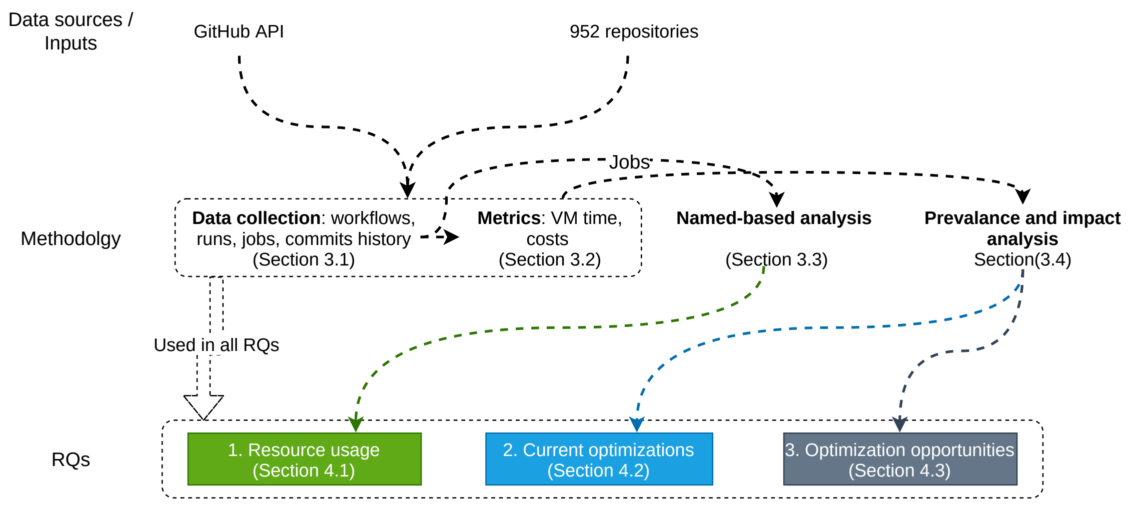
Data sources / Inputs
GitHub API
952 repositories
Jobs
Methodolgy
Data collection: workflows, runs, jobs, commits history (Section 3.1)
Metrics: VM time, costs (Section 3.2)
Named-based analysis (Section 3.3)
Prevalance and impact analysis Section(3.4)
Used in all RQs
RQs
1. Resource usage (Section 4.1)
2. Current optimizations (Section 4.2)
3. Optimization opportunities (Section 4.3)

Figure 1: Overview of the methodology we follow to answer the presented research questions.

job ID, the job status (whether it succeeded or failed), the job start and end times, and other metadata.

Applying our methodology, we collect detailed information about workflow runs of 952 repositories. The repositories contain code written in 14 different languages with Javascript/Typescript being the most popular (used in 16.2% of repositories), followed by Python (14.9%), Java, and Go (6.2% each). In total, there are 1.3 million workflow runs triggered by 20 different trigger events. All runs have executed between September 2020 and February 2023. The workflows executed a total of 3.7 million jobs and 34.3 million steps. As our dataset is the largest of its kind, we make it publicly available to foster future research.[2]

3.2 Metrics of Resource Usage

As a basis for characterizing the computational and monetary cost of executing workflows, we define and compute several metrics.

VM time. To capture the wall-clock time of each executed job, we compute *VM time* as the time between the start and the termination of a job, which equals the time that the VM executing the job has been up and running for this purpose. Because each VM executes only a single job at a time, this metric accurately captures the time that a job consumes. The VM time of a workflow run is the sum of the VM time of each job triggered within the workflow. Unless mentioned otherwise, we give VM time in minutes, while normalizing fractions to 60 seconds, i.e., 3.25 minutes means 3 minutes and 15 seconds.

VM cost. As an estimate of the monetary cost imposed by executing workflows, we compute *VM cost* as the amount of US dollars charged by GitHub according to its pricing policy. The base price per VM time is \$0.008. As different operating systems and hardware impose different costs, GitHub multiplies the consumed VM time with a factor f determined by the kind of VM a job runs on. Following current pricing,[3] the factor for running jobs on Windows and macOS machines is $f = 2$ and $f = 10$, respectively. GitHub rounds the number of minutes up to the nearest whole minute. Given a VM time of t, the VM cost hence is:

$$VM\ cost = \lceil t * f \rceil * 0.008$$

3.3 Name-Based Analysis of Jobs

To better understand the CI/CD tasks that developers want to perform with jobs in a workflow, we categorize jobs based on their names. As part of the workflow syntax, developers give an identifier to each job under the attribute "name". For example, typical names include "Test", "Build-image", and "Release(Ubuntu)". We unify names by transforming them into lower case and by removing details provided in parenthesis. Next, we select the most frequent names and manually simplify them into a more general name. For example, the name "build-image" would be simplified into "build", which captures the CI/CD task of building the code of a project. The set of simplified names includes: test, build, deploy, analyze, lint, linux, release, integrat(ion), sync, and update.

Given the set of common, simplified names of CI/CD tasks, we try to map each job name into one of the categories. Specifically, we consider a job to belong to one of the categories if the simplified name of the category is a substring of the job's name. For example, a job named "testlinux" belongs to the "test" category. Names that do not match any of the common categories, e.g., "graphqlv0.3", are left in a separate "other" category. Across our dataset, the name-based analysis of jobs categorizes 69.3% of all jobs into a category that is not "other".

[2]https://doi.org/10.5281/zenodo.8344575

[3]https://docs.github.com/en/billing/managing-billing-for-github-actions/about-billing-for-github-actions

3.4 Prevalence and Impact of Optimizations

GitHub Actions offers several mechanisms that can help optimize the resource consumption of workflows. We select six such mechanisms by going through the documentation of the various configuration options for workflows and by identifying those related to resource consumption. These studied optimizations are caching, fail-fast, canceling in-progress runs, skipping workflows, filtering by target files, and specifying a timeout (more details in Section 4.2).

To study how commonly these optimization-related mechanisms are used and what impact using them has on the consumed resources, we analyze the commit history of the studied repositories to identify commits that enable or disable a particular optimization. Specifically, we identify all commits that edit an already existing workflow file or create a new file. A workflow file is any file with a name that ends with "yaml" or "yml", and that is located under the folder ".github/workflows". Following that, we analyze the diff between the old version of the file (before commit) and the new version (after commit) to check whether an optimization was added to or removed from an already existing workflow, or whether a new workflow was created with an optimization enabled. This check is implemented through a regular expression specific to each optimization. Finally, we keep all those commits that add or remove an optimization, and refer to them as *optimization-related commits*.

To measure the prevalence of an optimization, we compute its *adoption rate* as the number of repositories with at least one optimization-related commit divided by the number of all studied repositories. To quantify the impact of using an optimization, we define three metrics. First, we compute the *percentage of affected runs*, which indicates how many of all runs of a workflow where the optimization is enabled are impacted by the optimization. For optimizations that affect all runs of a workflow, such as caching, this percentage is 100%. For optimizations that affect only some runs of workflow, such as specifying a timeout, this percentage is the number of runs where the optimization changes the VM time, e.g., because the workflow times out, divided by the number of workflow runs executed while the optimization is present in the workflow.

Second, we compute the *impact on VM time*, which indicates how the overall VM time consumed by a workflow where an optimization is enabled changes due to the optimization. For optimizations that affect all runs of a workflow, the impact is computed by comparing the VM time t_{no_opt} of a run without the optimization, e.g., just before the optimization-related commit, with the VM time t_{opt} of a run with the optimization, e.g., just after the optimization-related commit. Given these two times, the impact on VM time is $\frac{t_{opt}-t_{no\_opt}}{t_{no\_opt}}$. For optimizations that affect only some runs of a workflow, we compare two amounts of VM time. On the one hand, the time $t_{consumed}$ consumed by all runs of the workflow where the optimization is enabled. On the other hand, the time t_{saved} of time saved due to the optimization (described in detail in Section 4.2). Given these two times, the impact on VM time is $-\frac{t_{saved}}{t_{consumed}+t_{saved}}$.

Third, we calculate the *impact on annual cost per repository*, which signifies the influence of enabling an optimization on the financial burden incurred by a workflow. To this end, we consider the VM cost c_{opt} (Section 3.2) of runs for a workflow that has an optimization enabled. Additionally, we evaluate the VM cost c_{no_opt} that would have been incurred in the absence of the optimization, which is estimated based on the impact on VM time attributable to the optimization. The resulting difference, $c_{opt} - c_{no\_opt}$, is then divided over the number of years between the first and last executed run while the optimization is enabled, to get the annual cost delta.

4 RESULTS

4.1 Analysis of Resource Usage (RQ1)

We start off by gaining an understanding of resources consumed by workflows and what these resources are used for. In particular, we study the overall resource consumption, what events trigger resource usages, what CI/CD tasks are associated with the executed workflows, and how workflows typically terminate.

4.1.1 Overall Resource Usage and Costs. Overall, the 952 studied repositories use a total of 23.7 million minutes of VM time, with an average of 1,614 monthly minutes per repository. Each VM runs a specific operating system with Linux being the most used system, accounting for 64.6% of all jobs.

75.1% of repositories consume at most 2,000 minutes of VM time per month, which is within the free-tier limit, whereas the remaining 24.9% consume more than 2,000 minutes, which we refer to as paid-tier repositories. The paid-tier repositories consume an average of 5,914 minutes of VM time per month, while the free-tier have an average of 87 minutes per month, with an inter-quartile range of 3,490 and 93 minutes, respectively. Furthermore, 9.9 million runs belong to the paid tier, which represents 75.4% of the total runs in our dataset and 96.4% of the total VM time.

Looking at the costs, the yearly estimated VM cost of the paid-tier repositories, excluding the free minutes, is $504 (on average, inter-quartile range: $282). If repositories would have to pay for the free minutes as well, then a paid-tier repository would have to pay an extra $107 and a free-tier repository would have to pay $9, on average per year.

Finding 1: Running workflows cost a paid-tier repository around $504 per year, on average. GitHub covers $107 and $9 of the yearly expenses of paid-tier and free-tier repositories, respectively.

The key take-away of these findings is that workflows impose significant costs. While the yearly cost of a single repository seems manageable, the entire GitHub ecosystem consists of 386 million repositories, out of which at least 43 million are public.[4] Even though not all these repositories use workflows and our studied sample of repositories may not be fully representative, there most likely are many millions of repositories with workflows. As the costs of workflows are, in parts, covered by GitHub and by the respective project owners, both actors have a strong interest in understanding and reducing their respective share of the costs.

4.1.2 Events that Trigger Workflows. A workflow can be initiated by at least one of 36 distinct events,[5] of which 20 are observed within our dataset. Table 1 presents the list of events that consume

[4]According to https://github.com/search as of March 2023.
[5]https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows

Table 1: Summary of resource usage by triggering event.

Event	Overall (%)				Avg. per run			
	VM time		Runs		VM time (min) *		VM cost ($)	
	Paid	Free	Paid	Free	Paid	Free	Paid	Free
Pull Request	50.7	35.5	38.6	25.3	31.1 (20.1)	3.6 (3.6)	0.36	0.04
Push	30.9	47.8	26.4	28.6	28.4 (19.5)	4.3 (4.2)	0.33	0.05
Schedule	15.5	14.5	26.2	40.3	13.8 (1.3)	0.9 (0.2)	0.17	0.01
PR target	1.2	0.6	4.2	1.4	8.5 (11.9)	1.2 (1.3)	0.08	0.01
Dispatch	0.7	0.5	0.2	0.3	71.9 (24.9)	5.1 (4.4)	0.87	0.06
Workflow run	0.7	0.0	0.7	0.4	23.2 (13.0)	0.1 (0.1)	0.17	<0.01
Release	0.2	0.5	0.1	0.3	40.0 (15.0)	4.2 (5.9)	0.30	0.04
Others	0.1	0.6	3.6	3.4	4.5 (1.4)	0.7 (0.6)	0.04	0.01

* mean (inter-quartile range)

most VM time, as well as the proportion of total VM time consumed and the proportion of the total number of workflow runs. We also present a comparison between the paid tier and the free tier.

Pull requests, pushes, and scheduled workflows emerge as the most prevalent triggering events, accounting for 91.3% of the workflows of the paid tier and 94.4% for the free tier. Moreover, workflows initiated by these top three events represent 97.1% and 97.6% of the total VM time across all workflows of the paid and free tier, respectively. We also notice that the frequency of pull requests is higher among paid-tier repositories (38.6%) compared to the free-tier (25.3%). Furthermore, half of the VM time of the free tier is consumed by push events, contrary to the paid tier, where half of the VM time is consumed by pull requests. We attribute this phenomenon to differences in size and popularity: Paid-tier repositories tend to be larger and more popular, i.e., they have more collaborators, and hence, more activity through pull requests. In our dataset, the average number of collaborators triggering a workflow in a paid-tier repository is 22 compared to 12 per free-tier repository.

The average runtime of a workflow varies depending on the specific triggering event, primarily due to the differing jobs and tasks executed by each event. Remarkably, the longest running workflows are workflows triggered by a dispatch event, which means that the developer triggered the workflow manually either through the browser interface, GitHub CLI, or GitHub REST API. Using name-based analysis, we observe that the top tasks executed in dispatched workflows are build, test, and release.

Depending on the triggering event, paid-tier workflows take 7–100x longer than free-tier workflows. Of course, a larger average runtime directly correlates with larger monetary costs. For example, pull requests in paid-tier repositories cost 36 cents, on average, whereas they cost only 4 cents in free-tier repositories. This finding motivates paid-tier repositories to optimize their workflows, with the goal of lowering costs and increasing development efficiency.

Finding 2: 97.1% of the total VM time in our dataset is consumed by the top three triggers: push, pull request, and scheduled workflows. Paid-tier repositories have a higher frequency of pull requests, and they take 7–100x longer to execute a workflow compared to the free tier.

Table 2: Summary of resource consumption by CI/CD tasks.

Task	Overall (%)				Avg. per run			
	VM time		Runs		VM time (min) *		VM cost ($)	
	Paid	Free	Paid	Free	Paid	Free	Paid	Free
Test	54.6	35.9	51.1	33.8	8.0 (7.2)	1.6 (1.4)	0.10	0.02
Build	36.6	51.8	28.4	52.1	9.6 (8.3)	1.5 (1.2)	0.12	0.02
Release	3.5	1.0	2.4	1.8	10.0 (19.9)	0.8 (0.7)	0.08	0.01
Analyze	1.9	6.8	2.1	3.1	6.6 (5.6)	3.4 (2.4)	0.08	0.04
Lint	1.0	2.5	5.3	4.9	1.8 (1.7)	0.8 (0.5)	0.02	0.00
Linux	0.9	0.4	1.5	0.4	4.5 (1.9)	1.4 (1.6)	0.05	0.02
Update	0.7	0.2	5.7	0.5	1.0 (1.0)	0.7 (1.2)	0.01	0.01
Integration	0.4	0.8	1.2	0.5	2.6 (2.1)	2.4 (1.4)	0.03	0.03
Deploy	0.3	0.5	1.6	2.2	1.2 (1.5)	0.3 (0.2)	0.01	0.00
Sync	0.1	0.1	1.7	0.8	0.2 (0.0)	0.2 (0.1)	0.00	0.00

* mean (inter-quartile range)

4.1.3 *CI/CD Tasks and their Costs.*

To understand which CI/CD tasks account for resource usage, we classify the jobs executed as part of workflows using the name-based analysis described in Section 3.3. Table 2 shows the results. Among the jobs covered by the name-based analysis, testing and building are the dominant tasks both in terms of VM time and number of runs. On average, executing a testing job costs 10 cents, while a building job costs 12 cents. Releases and analyzing the code are also relatively time-expensive tasks, but are performed less frequently (less than 3% each).[6] Table 2 also shows that paid-tier repositories have a higher testing frequency compared to building, while for free-tier repositories building is more frequent. One possible explanation is that free-tier repositories trigger 30.2% of builds in scheduled workflows, which leads to higher, and potentially unnecessary, frequency of builds.

Finding 3: The most commonly executed CI/CD tasks are testing and building, which together consume around 90% of the total VM time among the top 10 studied tasks.

Our findings indicate that testing and building are critical tasks in workflows, and that optimizing these tasks could result in significant reductions of the consumed computing resources. Developers hence can benefit from techniques that optimize CI/CD pipelines [21, 22]. In addition, developers could use tools to reduce the size of the test suite, while keeping the same code coverage to help optimize the testing time [19, 31].

4.1.4 *Termination Status of Workflows.*

We also study the termination status of workflows, as it may hint at possibilities for optimizations. Workflows can terminate with one of the following conclusions: success, failure, skip, canceled, startup failure, action required, or stale. Table 3 presents the proportion of workflow runs and VM time corresponding to each conclusion. While the majority of workflows for both the paid tier (78.8%) and the free tier (88.9%) succeed, i.e., all jobs and steps complete successfully, a non-negligible fraction of workflows fails (17.4% and 10%, respectively).

[6] Our manual inspection shows that the name "Linux" is usually used with tests or builds when they are meant to be on Linux. For example, a workflow file might have the name tests.yml, while the jobs names are: Linux, Windows-10, and Mac-OS, referring to testing on these platforms.

Table 3: Termination status: comparison between free tier and paid tier.

Status	Runs proportion %		VM time proportion %	
	Paid	Free	Paid	Free
Success	78.7	88.9	66.3	81.1
Failure	17.4	10.0	30.9	18.0
Skipped	2.2	0.6	0.0	0.0
Canceled	1.5	0.3	2.7	0.8
Startup failure	0.1	0.1	0.0	0.0
Action required	< 0.1	0.1	0.0	0.0
Stale	< 0.1	0.0	0.0	0.0

Surprisingly, there appears to be a disparity in success percentages between free-tier and paid-tier repositories. The difference can be attributed to the lower complexity of free-tier repositories, which results in simpler tests and builds with fewer errors. A detailed inspection of our data, however, reveals that the free-tier repositories have a lower frequency of contributions (in the form of push or pull requests). Furthermore, our findings show that the activated workflows, at some point of time, in these repositories are primarily scheduled workflows, which account for a significant proportion (40.3%) and predominantly end with success due to rare or slow evolution of the repository.

The table also shows how much VM time is spent on workflows with a particular termination status. Interestingly, workflows that end in failure impose a disproportionately high computational cost, which motivates some of the optimizations studied in RQ2 and RQ3. A workflow is considered failed even if only a single job within it fails, resulting in the inclusion of the execution time of successful jobs in the failure time calculation. Furthermore, certain scenarios exist where the cancellation of a job leads to the termination of the workflow as a failure rather than a mere cancellation event. For instance, when a job exceeds its timeout, it is canceled, but the corresponding workflow run is classified as failed, thereby incorporating the canceled job's time into the overall failure time. The average VM time for successful jobs is 7.1 minutes, while it is 7.4 minutes for failed jobs, and 19.0 minutes for canceled jobs.

Finding 4: Due to less evolution and repeated successful scheduled workflows, free-tier repositories have a higher rate of successful workflows. Timed-out jobs cause a spike to the overall time of failed runs.

Our findings provide empirical motivation for work on predicting job failures and on prioritizing those jobs that are most likely to fail [21, 30]. Furthermore, optimizing failing scheduled workflows, as well as timed-out workflows, emerges as an intriguing optimization target. For instance, we observed a specific repository with a scheduled workflow set to execute every 5 minutes, which has consistently experienced failures over the course of the past months, amounting to a staggering 12,000 consecutive failures. In RQ3, we will delve into the discussion of optimization techniques specifically tailored for addressing failing jobs, such as timed-out workflows and other types of failures.

4.2 Current Optimizations and their Effectiveness (RQ2)

The following addresses the questions to what extent workflows use currently available optimizations, and how these optimizations impact the consumed resources. We study the prevalence and impact of six optimizations, summarized in Table 4.

4.2.1 Caching. Usually, each job starts with a clean runner image and then has to download any dependencies and other data as part of the job's execution. This process results in increased network utilization, longer runtime, and higher costs. To avoid repeatedly creating the same state, workflows can use a caching mechanism, which allows files to be reused across multiple runners. This optimization is provided as an action, which has been continuously evolving, currently encompassing versions 1 through 3.

Measuring saved time. We compute the time savings achieved by workflows that use caching by comparing the VM time t_{opt} of every caching-enabled run to the VM time t_{no_opt} of another run of the same workflow. Specifically, t_{no_opt} is for the closest-in-time run of the workflow that does not use caching. The difference $t_{no\_opt} - t_{opt}$ between these two runs represents the saved time. In some cases, the difference may be negative, indicating that incorporating caching does not have the intended effect. To isolate the impact of other changes introduced to the repository, we select changes where only the cache action is modified, which represents 51.3% of the total cases.

Prevalence and Impact. 32.9% of paid-tier repositories and 17.8% of the free-tier use caching at least once in their workflows, which reduces the execution time by an average of 3.4% and 6.0%, respectively. The saved time leads to a reduction in annual costs by $21.48 for the paid tier.

Take-away. The cache action is easy to use as it only requires adding one line to the workflow file to call the action. Given its ease of use and its impact on VM time, we believe that the cache option should be used more often, especially by paid-tier repositories.

4.2.2 Fail-Fast. This optimization applies to workflows that define an entire matrix of jobs, which is used, e.g., to run similar jobs on different operating systems. If fail-fast is enabled, then the platform will cancel all in-progress and queued jobs in the matrix as soon as any job in the matrix fails.

Measuring saved time. The saved time is measured by first selecting all those runs with a matrix of jobs where one of the jobs ended in failure, which causes the other runs to be canceled. We call the time taken by such a workflow $t_{canceled}$. As a reference for comparison, we identify for each such run the closest-in-time, non-failing run of the same workflow, and we call its execution time $t_{succeeded}$. The difference $t_{succeeded} - t_{canceled}$ between the two runs is the saved time.

Prevalence and impact. This option is slightly more adopted by free-tier (83.5%) than paid-tier (75.9%) repositories. The high adoption rate is likely due to the fact that fail-fast is enabled by default. This option reduces VM time by 1.5% and 2.0% for paid-tier and free-tier repositories, respectively.

Table 4: Prevalence and impact of workflows optimizations.

Optimization	Default	Adoption rate (%)		Impacted runs (%)		Impact on VM time (%)		Annual cost delta / repo ($)	
		Paid	Free	Paid	Free	Paid	Free	Paid	Free
Cache	Off	32.9	17.8	100.0	100.0	-3.4	-6.0	-21.48	-0.59
Fail-fast	On	75.9	83.5	3.1	4.7	-1.5	-2.0	-2.13	-4.21
Cancel-in-progress	Off	10.1	4.6	9.1	1.5	-4.1	-1.9	-55.14	-0.43
Skip workflow	–	9.7	4.9	0.1	0.3	<-0.1	-0.4	-2.52	-0.75
Filtering target files	Off	20.7	8.7	<0.1	<0.1	<-0.1	<-0.1	-2.20	-0.06
Custom timeout	360 mins	14.0	2.6	4.3	1.7	-8.1	-12.9	-58.33	-1.58

Take-away. Although the fail-fast option is activated in a majority of repositories, its impact on VM time is relatively low, resulting in a few dollars in annual savings per repository. Nevertheless, we posit that enabling such an option by default is beneficial, and GitHub could potentially identify other options to activate by default. This approach would promote efficiency and encourage developers to adopt these optimizations.

4.2.3 Cancel-In-Progress. This optimization allows developers to automatically cancel an in-progress run of a job if a new run of the same job is triggered in the meanwhile.

Measuring saved time. To quantify the saved time, we consider all runs of workflows that use the cancel-in-progress option and that are canceled. Let the VM time of such a run be $t_{canceled}$. Next, for each such run, we check whether another run of the same workflow starts within a 5-second interval following the cancellation of the first run. Let the time taken by the second run be t_{next}. Finally, the time difference $t_{next} - t_{canceled}$ between the newly triggered run and the canceled run is the saved time.

Prevalence and impact. This optimization is used by 10.1% of paid-tier repositories and only 4.6% of the free-tier ones. Within the paid tier, Cancel-in-progress impacts 9.1% of runs, reduces VM time by 4.1%, and saves $55.14 yearly per repository.

Take-away. This optimization is not widely invoked among the paid-tier repositories. One explanation is that the vast majority of these repositories accept pull requests from other developers. Cancelling an ongoing workflow triggered by a pull request becomes impractical when subsequent pull requests are submitted, as each workflow is designed to assess the specific state of the repository branch associated with the pull request submitter, e.g., via testing. Additionally, this optimization option solely impacts workflow runs triggered by push and pull requests, meaning that even if invoked, it cannot be utilized in 35.0% of the runs.

4.2.4 Skip Workflow. Workflows that are triggered by repository pushes and pull requests are executed for each of these events. By adding any of the following strings to a commit message, developers can skip the execution of all workflows: [skip ci], [ci skip], [no ci], [skip actions], [actions skip].

Measuring saved time. For commits with a "skip workflow" note, we estimate the saved time as the VM time of the run closest in time to the skipped one.

Prevalence and impact. 9.7% of paid-tier repositories have used this option in at least one of their commit messages. The option to skip workflows is used in very few commits and thus impacts only a small number of runs.

Take-away. The infrequent use of this option results in only limited time savings. One explanation for its sparse usage could be the lack of awareness surrounding the availability of such an option. To address this issue, GitHub could implement a prompt message via their GitHub CLI or GitHub Desktop app, which would inquire whether users want to trigger workflows upon pushing a commit. This approach would not only enhance the practicality of the option but also raise awareness of its existence.

4.2.5 Filtering Target Files. Developers can specify whether a workflow runs based on the kind of file edited as part of an event that triggers the workflow, e.g., the files edited in a commit. The target file can be specified via a white list or a black list of files.

Measuring saved time. When all of the files changed by a commit are in the black list of a workflow and none are in the white list, the workflow is marked as skipped. This information is used to select skipped workflows during the time interval where path filtering is used. We consider the saved time as the total VM time of the run that is temporally closest to the run that was skipped for each skipped run of the same workflow.

Prevalence and impact. This option is adopted by 20.7% of paid-tier repositories. However, it leads to only $2.20 of yearly savings.

Take-away. Although this option is used in roughly one-fifth of paid-tier repositories, the affected runs and their impact on VM time remain minor. This option only saves time when all files modified by a commit are filtered out, which happens infrequently. These results will help developers gain insights on how beneficial this optimization is overall.

4.2.6 Custom Timeout. Instead of relying on the default timeout of jobs, which is 360 minutes (6 hours), workflows can explicitly set a custom timeout. Repositories may use this option to stop workflows from running unnecessarily long, which may be particularly useful to prevent contributors, e.g., via pull requests, to trigger long-running workflows.

Measuring saved time. The saved time for each timed-out job is calculated as the difference between the timeout threshold value and the longest VM time the timed-out job has ever taken. The reasoning behind this measurement is that if the timeout threshold is not used, the job may execute for a longer period of time, and the maximum VM time a job has ever reached serves as an estimate of how long that job would run.

Prevalence and impact. 14.0% of the paid-tier repositories use a custom timeout at some point, which impacts 4.3% of runs and leads to a decrease of 8.1% in VM time, equivalent to saving $58.33 yearly per repository. While only 2.6% of the free-tier repositories use this option, it reduces VM time by 12.9%.

Take-away. Setting a timeout for jobs is a powerful option and should be used more often. However, developers lack direct tools that help them set the timeout to a value that maximizes their saving while keeping the risk of interrupting a normally lasting job low. We hypothesize that artificial intelligence techniques could be used to predict a near-optimal timeout value for a job based on the job itself and the history of the repository, which future work should explore further.

Finding 5: Paid-tier repositories are generally more likely to adopt optimization mechanisms (except for fail-fast), which shows the importance of optimizations to those repositories. The studied mechanisms save up to $60, on average per year and repository.

4.3 Optimization Opportunities (RQ3)

The following presents four optimization mechanisms designed to reduce the amount of time and resources consumed by executed workflows. Currently, these mechanisms are not present in the proposed form on GitHub Actions, but our results suggest that they could significantly reduce both computational and financial costs. Some of the discussed mechanisms resemble optimizations proposed for regression testing and for other CI/CD platforms (Section 7), and we do not claim novelty of the mechanisms. Instead, this research question quantifies the extent to which the optimization mechanisms would affect workflows and estimates their impact on costs. To this end, we estimate the impact of each mechanism had it been implemented for the workflow runs in our dataset. Table 5 summarizes our main results. We focus mainly on the paid-tier repositories. Nevertheless, we also report the impact on free-tier repositories given between parenthesis in each row of the table.

4.3.1 Deactivate Scheduled Workflows After k Consecutive Failures. This mechanism aims to deactivate any scheduled workflow that, based on its previous runs, is likely to fail. In our dataset, scheduled workflows account for 29.8% of all runs and consume 15.4% of total VM time. Notably, 13% of these scheduled runs result in failure. Inspecting the sequence of failing scheduled workflows over time in our dataset reveals that numerous scheduled workflows continue to execute and fail consecutively before developers intervene. To address this issue, we propose an optimization mechanism that deactivates a scheduled workflow after k consecutive failures, and sends a notification to the developers. The rationale behind this approach is to prevent further resource wastage until developers examine and address the cause of the failures.

A hyperparameter of this mechanism is the number k of consecutive failures after which the mechanism deactivates the workflow. With k=3, the optimization impacts 4.5% of the paid-tier runs, and in particular, it impacts 17.2% of scheduled runs, which saves 3.2% of the total VM time of all runs and 21.3% of VM time of all scheduled runs of the paid tier. The saved time corresponds to yearly savings of $125.72 per impacted repository, on average. Setting the parameter k to higher values decreases the proportion of impacted runs and hence the saved time as shown in Table 6.

4.3.2 Deactivate Scheduled Workflows During Repository Inactivity. Scheduled workflows usually perform a task related to the current state of the repository. For example, a scheduled workflow may build and release the software every day. However, by default a scheduled workflow continuously runs even if the state of the repository does not change at all, which wastes resources. To overcome this issue, we propose an optimization mechanism that prevents a scheduled workflow from running if there was no activity between the previous, already executed scheduled run and the scheduled run that is about to start. Currently, GitHub Actions deactivates scheduled workflows after six months of inactivity, which may lead to many unnecessary workflow runs in the meanwhile.

Evaluating this mechanism on our dataset shows that it affects 4.3% of paid-tier runs, and in particular, 16.3% of the scheduled workflows. Deactivating these workflows while their repositories do not change leads to a decrease of 1.1% in VM time of scheduled workflows . The saved time translates to yearly savings of $10.41 per impacted repository, on average. Unfortunately, some scheduled workflows write changes into the repository (e.g., change build log) at the end of the workflow execution, which would appear as an activity in the repository. An improvement over our heuristic is to ignore changes made by the scheduled workflow itself.

4.3.3 Run Previously Failed Jobs First. This optimization mechanism addresses workflows that execute multiple jobs, e.g., by running a test suite on different operating systems. We observe that when a job in such a workflow fails, the same job sometimes also fails in the subsequent run of the workflow. The idea of the optimization is to reorder jobs within a workflow to prioritize the execution of previously failed jobs. If a previously failed job fails again, the reordering will prevent the other, previously succeeding jobs from being executed a second time, which saves the VM time they would consume. If, instead, the previously failed job succeeds, then all other jobs will be run as well, and there will be no saved VM time.

Implementing this mechanism affects 1% of all runs, and in particular, 29.8% of failed runs of paid-tier repositories. The optimization results in a 1.1% reduction in VM time across all runs and in a 31.6% reduction for failed runs. The saved time translates to yearly savings of $17.89 per impacted repository, on average. Besides reducing the computational and monetary cost of workflows, running previously failed jobs first could also speed up the development process. The reason is that the developers will potentially see a failure faster, and hence, can react to it earlier.

4.3.4 Job-Specific Timeouts. Jobs run as part of a workflow may take longer than expected, and in the worst case, even fail to terminate, e.g., due to an infinite loop or because of waiting for an external event. By default, the GitHub Actions platform terminates jobs after a timeout of six hours, which is a generous value given

Table 5: Prevalence and impact of our suggested optimization techniques in paid tier (free tier).

Optimization heuristic	Impacted runs *	Time saving *	Annual cost delta per repository in $ *
Deactivate scheduled workflows after k consecutive failures (k=3)	4.5% (<0.1%) of all runs 17.2% (1.0%) of scheduled runs	3.2% (<0.1%) of all runs time 21.3% (4.5%) of scheduled runs time	-125.72 (-1.55)
Deactivate scheduled workflows during repository inactivity	4.3% (0.2%) of all runs 16.3% (0.5%) of scheduled runs	<0.1% (<0.1%) of all runs time 1.1% (<0.1%) of scheduled runs time	-10.41 (-7.97)
Run previously failed jobs first	1.0% (0.8%) of all runs 29.8% (7.7%) of failed runs	1.1% (<0.1%) of all runs time 31.6% (45.3%) of failed runs time	-17.89 (-0.77)
Project-specific timeouts	0.5% (<0.1%) of all runs	3.5% (2.2%) of all runs time	-173.71 (-47.49)

* measurement for paid tier (measurement for free tier)

Table 6: Sensitivity of deactivating scheduled workflows to the parameter k.

k	1	2	5	10	15	20
Impact on VM time (%)	-3.8	-3.4	-2.8	-2.3	-1.9	-1.6

that, on average, jobs are running for only 6.3 minutes. Instead, we assess the impact of imposing a job-specific timeout set to be sufficient for all runs of a job that finish within the current six-hour timeout.

Specifically, we consider for each job the maximum time t_{max} that a non-timed-out run of the job has ever consumed, compute a timeout value that adds a 10% buffer on top of this value, i.e., $timeout = 1.1 * t_{max}$, and then assume the job-specific timeout of each job to be $min(timeout, 6h)$. While we determine a suitable job-specific timeout for the jobs in our dataset after the fact, a practical implementation of this optimization mechanism could estimate the timeout based on past runs of a job. To estimate the impact of the optimization, we consider every run of a job in our dataset that exceeds the 6-hour default limit. The difference between our job-specific timeout and the 6-hour default is VM time saved by the optimization.

Implementing this mechanism would affect only 0.5% of all runs but would yield a 3.5% reduction in total VM time. The reason for this relatively large difference is that unnecessarily waiting until the 6-hour default timeout occurs can easily consume many VM minutes. On average per repository, the optimization leads to a yearly cost saving of $173.71.

Finding 6: Estimating the impact of four currently missing optimization mechanisms shows a savings potential up to 3.5% of the total VM time observed in our dataset.

In general, our results show that the proposed techniques have most impact both on the number of runs and on VM time when applied to paid-tier repositories. By implementing these suggested mechanisms, developers and the GitHub platform could benefit from reduced resource consumption, cost savings, and improved CI/CD pipeline performance. Our empirical results offer guidance on which mechanisms are the most promising. We believe that integrating these heuristics into the GitHub Actions ecosystem would be relatively straightforward due to the simplicity of the optimizations. Reducing the cost imposed by workflows will help GitHub to maintain its commitment to providing free resources to open-source projects, while reducing its costs, and also makes the platform more attractive to paying users.

5 DISCUSSION

Costs and impact of GitHub Actions. The growing popularity of GitHub Actions, combined with the computational demands of CI/CD tasks, results in substantial costs. This is the first study to quantify these costs, taking into account both their monetary and temporal aspects. Raising awareness of CI/CD costs is relevant to both platform providers, e.g., GitHub, which provides significant resources for free to public repositories, and for developers with non-free accounts. Our findings can help both to better understand resource consumption and how to reduce it. Reducing the resources consumed by CI/CD could also have a positive environmental impact because computation contributes significantly to global carbon emissions [5, 33].

Efficiency and optimization techniques. Our study performs an in-depth analysis of the optimization-related configuration options offered by GitHub that could lead to reduced VM time. The currently low adoption rate of some optimizations suggests that better documentation and tool support could guide developers toward more resource-efficient workflows. For example, activating some optimizations by default or suggesting the best configuration set based on the task (test, build, etc.) are promising directions. Such enhancements could help reduce resource waste and increase software development efficiency. In addition, our study quantifies the impact of these optimizations on VM time, giving developers an idea about the most effective techniques. It would be interesting to see similar measurements provided by GitHub on a larger scale.

Motivated by the room for improvement, we suggest and empirically evaluate optimization mechanisms that are currently not present on GitHub Actions. These mechanisms promise a considerable impact, which could complement existing optimizations. It would be interesting to study how developers react and use these techniques in practice.

6 THREATS TO VALIDITY AND LIMITATIONS

Our study is exposed to several factors that could limit the generalizability and validity of the findings. To begin, our analysis is based on a dataset of 952 repositories, which is a small subset of the hundreds of millions of repositories on GitHub. As a result, our sample may not be representative of all repositories. In addition, our research is limited to GitHub Actions, which is only one of many CI/CD platforms. It would be interesting to investigate how our results about GitHub compares to other platforms. Next, our monetary cost estimate is based on GitHub's pricing model, which most likely includes not only the actual cost but also a margin of profit. Thus, the cost estimates presented in this study may be slightly inflated. Moreover, our name-based analysis of jobs and their corresponding CI/CD tasks may not be completely accurate. In particular, our methodology leaves the task of 30.7% of jobs unclassified, which may have an impact on our overall understanding of resource utilization patterns.

Our results on the benefits of optimizations are estimates, and hence, may not be fully accurate. For example, reordering jobs, as proposed when running previously failed jobs first, may not be possible when jobs dependent on one another. However, our analysis shows that only 10.3% of workflows implement such dependencies, and we exclude such workflows. When studying fail-fast usage, we consider only jobs in a matrix without considering concurrency groups where fail-fast is also applicable. In our dataset, 3.5% of workflows use concurrency groups, which we ignore. Finally, in the context of RQ2, with the exception of caching, estimating the expected VM time relies on comparing a workflow run with the temporally closest run, which ignores any changes committed to the repository between these two runs. However, analyzing the time difference between consecutive workflow runs in our dataset, we find that, on average, there is a 0.6% increase in the VM time (with an interquartile range of 1.0%). While this effect is relatively small, it remains a potential threat to validity, as the specific value varies across workflows.

Finally, because the hardware that workflows are executed on is unknown to us, our study does not account for potential differences in hardware specifications. Instead, all our results assume the minimum hardware configuration. This limitation may cause some variation in our estimates of resource usage and optimization opportunities. In addition, users also pay for used storage. Unfortunately, the GitHub API currently does not give information about storage usage, and hence, we cannot study it here.

7 RELATED WORK

Prevalence of CI/CD. Several studies show the prevalence and importance of CI/CD. For example, Hilton et al. [18] reported already in 2016 that 70% of the most popular projects and 40% of all GitHub projects use continuous integration. A study of tests run during CI at Google shows that on an average day, 800k builds and 150M test runs are performed [29]. The TravisTorrent dataset includes logs of millions of Travis CI builds [3], which has enabled studies of the Travis CI platform, e.g., on build failures [2].

Empirical studies on GitHub Actions and other CI platforms. Since its inception in 2019, GitHub Actions has been established as a major CI/CD platform. For example, a recent study reports that 44% of the studied projects are using GitHub Actions [9]. Other studies confirm the increasing adoption of GitHub Actions and show how using the platform affects development, e.g., measured in terms of merged pull requests [7, 23, 39]. There also are studies on security issues related to workflows [4, 24]. Our work fundamentally differs from the existing studies by providing the first in-depth study of resource usage and optimizations. Beyond GitHub Actions, other CI platforms have been studied, e.g., CircleCI [13]. Comparing the resource usage patterns of across different platforms would be interesting future work.

Bad practices in CI/CD. Some of our results point out suboptimal practices in using the resources offered along with GitHub Actions, which relates to prior work on studying [34] and detecting [38] bad practices in CI/CD. For example, Gallaba and McIntosh [14] propose automated detection of anti-patterns in Travis CI specifications, such as misspelled property names and running commands unrelated to the current CI phase. A study shows that unhealthy CI practices are relatively common, e.g. infrequent commits, running test suites despite poor test coverage, and taking long to fix a broken build [11]. Work by Zampetti et al. [40] reveals bad practices via developer interviews and mining Stack Overflow, e.g., mismanagement of branches and incorrect versioning. None of the above work is on resource usages and bad practices related to them, which is the contribution of this paper. Vassallo et al. report that many developers consider slow builds to be problematic [37], which, as shown in RQ1, is an important contribution to the overall time required by workflows.

Improving CI/CD. Various techniques for improving CI/CD have been proposed. One line of work is about predicting the outcome of a build, which may give developers feedback faster than waiting for the actual build outcome [16, 35]. A related idea is about predicting the outcome of builds [6, 20, 22] and test suite executions [30] in order to skip unnecessary builds. This ideas relates to some of the optimization opportunities we point out in Section 4.3. We contribute by quantifying the cost savings that optimizations could provide.

Studying and optimizing regression testing. As regression testing accounts for a large part of the overall CI/CD costs (Section 4.1), various studies and techniques are about optimizing it. Labuschagne et al. [25] study how many failures are typically detected by regression testing and what reasons, e.g., flaky tests, cause these failures. To reduce the time taken by running regression tests, techniques for test selection and test prioritization [10, 27, 32], for test case failure prediction [1, 28], and for prioritizing commits for which regression testing should be performed first [26] have been proposed. Jin and Servant [21] compare different techniques for selecting and prioritizing tests and builds. All these techniques operate at a more fine-grained than job-level or workflow-level optimizations, as they aim at reducing the cost of a specific kind of job, and hence are orthogonal to the optimizations we study in this work. Kotinos [12] propose to accelerate CI builds by automatically inferring opportunities for caching and for skipping of builds, based on monitoring of system calls. Their work agrees with our finding that many projects could optimize their CI and shows a concrete technique to help reach this goal.

Supporting developers in using CI. To help developers in using CI/CD platforms, Ziftci and Reardon [41] propose a technique for finding the code change that caused a test failure. Other work helps by localizating and fixing build errors [36]. Finally, Hilton et al. [17] report on interviews with developers, which reveal the motivations for adopting CI systems and barriers developers facein this process. We envision the findings of our study to motivate future techniques that help developers in reducing the computational cost of CI.

8 CONCLUSION

This paper presents the first empirical study of resource utilization and optimization opportunities within the GitHub Actions platform, shedding light on the complexities of CI/CD processes and their associated costs. Our findings highlight the importance of implementing more efficient workflows and utilizing available optimization techniques, which reduces costs and helps speed up the development process. Furthermore, the findings call for improved documentation and tool support to guide developers toward resource-efficient practices. Overall, our work contributes to the development of a more sustainable and efficient software development ecosystem by identifying existing inefficiencies and proposing novel optimization heuristics.

DATA AVAILABILITY

Our code and data are available at: https://doi.org/10.5281/zenodo.8344575

ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and DeMoCo projects. We would also like to thank the reviewers for their valuable feedback that helped improve the paper.

REFERENCES

[1] Jeff Anderson, Saeed Salem, and Hyunsook Do. 2015. Striving for failure: an industrial case study about test failure prediction. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 49–58.

[2] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. In *International Conference on Mining Software Repositories.*

[3] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *IEEE/ACM International Conference on Mining Software Repositories.*

[4] Giacomo Benedetti, Luca Verderame, and Alessio Merlo. 2022. Automatic Security Assessment of GitHub Actions Workflows. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses.* ACM, Los Angeles CA USA, 37–45. https://doi.org/10.1145/3560835.3564554

[5] Rajkumar Buyya, Anton Beloglazov, and Jemal Abawajy. 2010. Energy-efficient management of data center resources for cloud computing: A vision, architectural elements, and open challenges. *arXiv preprint arXiv:1006.0308* (2010).

[6] Bihuan Chen, LinLin Chen, Chen Zhang, and Xin Peng. 2020. BuildFast: History-Aware Build Outcome Prediction for Fast Feedback and Reduced Cost in Continuous Integration. In *ASE.*

[7] Tingting Chen, Yang Zhang, Shu Chen, Tao Wang, and Yiwen Wu. 2021. 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).* IEEE, Hainan, China, 01–10. https://doi.org/10.1109/QRS-C55045.2021.00163

[8] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. *Sampling Projects in GitHub for MSR Studies.* Technical Report arXiv:2103.04682. arXiv. http://arxiv.org/abs/2103.04682 arXiv:2103.04682 [cs] type: article.

[9] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. 2022. On the Use of GitHub Actions in Software Development Repositories. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, Limassol, Cyprus, 235–245. https://doi.org/10.1109/ICSME55016.2022.00029

[10] Sebastian G. Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 235–245. https://doi.org/10.1145/2635868.2635910

[11] Wagner Felidré, Leonardo Furtado, Daniel A da Costa, Bruno Cartaxo, and Gustavo Pinto. 2019. Continuous integration theater. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).* IEEE, 1–10.

[12] Keheliya Gallaba, John Ewart, Yves Junqueira, and Shane McIntosh. 2022. Accelerating Continuous Integration by Caching Environments and Inferring Dependencies. *IEEE Trans. Software Eng.* 48, 6 (2022), 2040–2052. https://doi.org/10.1109/TSE.2020.3048335

[13] Keheliya Gallaba, Maxime Lamothe, and Shane McIntosh. 2022. Lessons from Eight Years of Operational Data from a Continuous Integration Service: An Exploratory Case Study of CircleCI. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 1330–1342. https://doi.org/10.1145/3510003.3510211

[14] Keheliya Gallaba and Shane McIntosh. 2018. Use and misuse of continuous integration features: An empirical study of projects that (mis) use Travis CI. *IEEE Transactions on Software Engineering* 46, 1 (2018), 33–50.

[15] Mehdi Golzadeh, Alexandre Decan, and Tom Mens. 2022. On the rise and fall of CI services in GitHub. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, Honolulu, HI, USA, 662–672. https://doi.org/10.1109/SANER53432.2022.00084

[16] Ahmed E. Hassan and Ken Zhang. 2006. Using Decision Trees to Predict the Certification Result of a Build. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06).* IEEE Computer Society, USA, 189–198. https://doi.org/10.1109/ASE.2006.72

[17] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* 197–207.

[18] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 426–437.

[19] Hwa-You Hsu and Alessandro Orso. 2009. MINTS: A general framework and tool for supporting test-suite minimization. In *2009 IEEE 31st international conference on software engineering.* IEEE, 419–429.

[20] Xianhao Jin and Francisco Servant. 2020. A Cost-efficient Approach to Building in Continuous Integration. In *ICSE.*

[21] Xianhao Jin and Francisco Servant. 2021. What helped, and what did not? An Evaluation of the Strategies to Improve Continuous Integration. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).

[22] Xianhao Jin and Francisco Servant. 2022. Which builds are really safe to skip? Maximizing failure observation for build selection in continuous integration. *Journal of Systems and Software* 188 (June 2022), 111292. https://doi.org/10.1016/j.jss.2022.111292

[23] Timothy Kinsman, Mairieli Wessel, Marco A. Gerosa, and Christoph Treude. 2021. How Do Software Developers Use GitHub Actions to Automate Their Workflows? (2021). https://doi.org/10.48550/ARXIV.2103.12224

[24] Igibek Koishybayev, Aleksandr Nahapetyan, Raima Zachariah, Siddharth Muralee, Bradley Reaves, Alexandros Kapravelos, and Aravind Machiry. 2022. Characterizing the Security of Github CI Workflows. In *31st USENIX Security Symposium (USENIX Security 22).* USENIX Association, Boston, MA, 2747–2763. https://www.usenix.org/conference/usenixsecurity22/presentation/koishybayev

[25] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the cost of regression testing in practice: A study of Java projects using continuous integration. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering.* 821–830.

[26] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining prioritization: continuous prioritization for continuous integration. In *Proceedings of the 40th International Conference on Software Engineering.* 688–698.

[27] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How does regression test prioritization perform in real-world software evolution?. In *Proceedings of the 38th International Conference on Software Engineering.* 535–546.

[28] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive test selection. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, Helen Sharp and Mike Whalen (Eds.). IEEE / ACM, 91–100. https://doi.org/10.1109/ICSE-SEIP.2019.00018

[29] Atif M. Memon, Zebao Gao, Bao N. Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing.

In *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE Computer Society, 233–242. https://doi.org/10.1109/ICSE-SEIP.2017.16

[30] Cong Pan and Michael Pradel. 2021. Continuous test suite failure prediction. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 553–565. https://doi.org/10.1145/3460319.3464840

[31] Saeed Parsa and Alireza Khalilian. 2010. On the optimization approach towards test suite minimization. *International Journal of Software Engineering and its applications* 4, 1 (2010), 15–28.

[32] David Paterson, José Campos, Rui Abreu, Gregory M Kapfhammer, Gordon Fraser, and Phil McMinn. 2019. An Empirical Study on the Use of Defect Prediction for Test Case Prioritization. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 346–357.

[33] Chris Preist, Daniel Schien, and Eli Blevis. 2016. Understanding and mitigating the effects of device and cloud service design decisions on the environmental footprint of digital infrastructure. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 1324–1337.

[34] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 164–175.

[35] T. Rausch, W. Hummer, P. Leitner, and S. Schulte. 2017. An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 345–355.

[36] Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. 2019. Learning to Fix Build Errors with Graph2Diff Neural Networks. (2019).

[37] Carmine Vassallo, Sebastian Proksch, Harald C. Gall, and Massimiliano Di Penta. 2019. Automated reporting of anti-patterns and decay in continuous integration. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 105–115. https://doi.org/10.1109/ICSE.2019.00028

[38] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C. Gall, and Massimiliano Di Penta. 2020. Configuration smells in continuous delivery pipelines: a linter and a six-month study on GitLab. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 327–337. https://doi.org/10.1145/3368089.3409709

[39] Mairieli Wessel, Joseph Vargovich, Marco A. Gerosa, and Christoph Treude. 2022. GitHub Actions: The Impact on the Pull Request Process. (2022). https://doi.org/10.48550/ARXIV.2206.14118

[40] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. 2020. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering* 25 (2020), 1095–1135.

[41] Celal Ziftci and Jim Reardon. 2017. Who Broke the Build? Automatically Identifying Changes That Induce Test Failures in Continuous Integration at Google Scale. In *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE Computer Society, 113–122. https://doi.org/10.1109/ICSE-SEIP.2017.13