On the Reruns of GitHub Actions Workflows

JIANGNAN HUANG, Radboud University, The Netherlands
BIN LIN*, Hangzhou Dianzi University, China

GitHub Actions, a built-in CI/CD solution of GitHub, is increasingly popular among developers for automating software development workflows. It has been observed that when automated workflow execution fails, developers sometimes only rerun the workflow or failed job without any modifications to the repository. The widespread use of these reruns has consumed considerable computing resources and raised concerns regarding the reliability and consistency of these workflows. Understanding how developers rerun GitHub Actions workflows and the rationale behind the rerun can provide valuable insights to further improve the reliability and efficiency of the software development process.

In this work, we conducted an empirical study on 3,320 open-source Java repositories to understand how developers rerun GitHub Actions workflows and quantify both wasted time and computing resources. We further studied the cases where workflow reruns lead to successful outcomes and manually analyzed the reasons behind the workflow execution flakiness. Based on our findings, we tested four machine learning based models to predict the workflow execution outcome, aiming to reduce the potential resource waste caused by workflow reruns. Our study presents how developers deal with GitHub Actions execution failures, pinpoints root causes of workflow execution flakiness, and offers actionable insights to improve CI/CD workflow reliability and efficiency.

CCS Concepts: • **Software and its engineering → Software configuration management and version control systems**; *Software libraries and repositories*; Software maintenance tools; Software development process management.

Additional Key Words and Phrases: GitHub Actions, CI/CD, Software Repositories

ACM Reference Format:
Jiangnan HUANG and Bin LIN. 2024. On the Reruns of GitHub Actions Workflows. *J. ACM* 37, 4, Article 111 (August 2024), 32 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

GitHub [25] is one of the largest social coding platforms, hosting over 420 million software repositories and more than 100 million developers as of June 2024. The constant evolution of GitHub including the integration of issue tracking and pull requests into its distributed version control system, has significantly changed the way how developers collaborate [7, 42]. Over the past decade, many third party continuous integration and delivery (CI/CD) tools have been created and adopted in GitHub projects to automate various development activities such as building, testing, and deployment [61]. In November 2019, GitHub publicly released GitHub Actions [26] (GHA, hereafter) as its own built-in solution for CI/CD. GHA is fully integrated into GitHub, and since its release, it has quickly gained popularity, replacing Travis as the dominant CI/CD tool within 18 months [41].

*Corresponding author.

Authors' Contact Information: Jiangnan HUANG, jiangnan.huang@ru.nl, Radboud University, Nijmegen, The Netherlands; Bin LIN, b.lin@hdu.edu.cn, Hangzhou Dianzi University, Hangzhou, China.

ACM 1557-735X/2024/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

Similar to other CI/CD tools, developers can specify automated tasks in a GHA workflow. Workflow files (WFs, hereafter) are the main components to configure GHA pipelines. Like ordinary source code, WFs are developed and modified throughout the project's lifetime to meet the needs of developers. In practice, not all the executions of GHA workflows are successful due to various issues. In instances of unsuccessful workflow executions, developers may need to troubleshoot the failures, update their repositories, and re-execute the workflow.

Typically, in CI/CD systems, not all the workflow execution failures are caused by code-related issues in the repository [45]. Instead, there are several other factors which might contribute to the failures, such as non-deterministic ("flaky") test results [10, 52, 53], infrastructure issues within testing environments [58], or outages with workflow service providers [11]. Suspecting that the source code is not responsible for a failed execution, the developers can rerun the workflow.

Recognizing that rerunning entire workflows can be both time and resource-intensive, GitHub introduced a new feature in 2022, allowing users to selectively rerun failed jobs or specific individual jobs in workflows within 30 days of the initial run [31]. While the reruns of workflows and jobs may be warranted if the execution outcomes turn into positive, indiscriminate reruns of workflows and jobs might unnecessarily consume computing resources. According to Sedano et al. [68], this kind of avoidable computation and corresponding waiting time constitute software waste as they do not provide significant values to software organizations.

Understanding the prevalence of GHA workflow reruns and the reasons behind it can provide actionable insights for reducing unnecessary reruns and optimizing resource management, thus leading to cost saving and further software development productivity improvement. Additionally, insights into rerun patterns can help better design GHA workflows, helping organizations to anticipate potential inefficiencies and adopt best practices for sustainable workflow management.

In this paper, we conducted an empirical study involving 9,202 WFs from 3,320 active Java projects from GitHub, to understand the common practice of GHA workflow reruns and assess the associated wastes. Our study presents an overview of the life cycle of workflow executions, and reveals the non-deterministic outcomes produced by workflow runs, namely the same workflow yields different execution results despite no changes made to the repository. While test flakiness plays a significant role in this issue, as suggested by previous studies [10, 52, 53], we discovered that other factors may also contribute to workflow execution flakiness. To gain a comprehensive picture of this matter, we conducted the first case study of flaky workflow executions. By manually analyzing the reasons behind the inconsistent outcomes of GHA workflow executions, we categorized the root causes of workflow flakiness. Furthermore, to minimize the waste of computational resources generated by reruns, we tested four different Machine Learning (ML) based models to predict the outcome of the workflow executions. Our best model demonstrates an F1-score of 0.9 in successfully predicting workflow execution outcomes. The main contributions of our study are as follows:

- Our study presents an overview of general life cycle of workflow executions, examines the adoption of the new GitHub job-rerun feature, and sheds light on the wastes generated by GHA workflow and job reruns.
- We categorize the reasons behind the flakiness of GHA workflow executions, and provide developers with valuable insights on how to eliminate or avoid such cases.
- We propose ML-based models to predict the outcome of GHA workflow executions, helping both developers and GitHub to save unnecessarily wasted time and computational resources.

2 BACKGROUND

A GHA workflow is defined as a YAML file, located in the designated `.github/workflows` directory of a GitHub repository, it consists of one or more *jobs* that execute certain tasks. Each job comprises

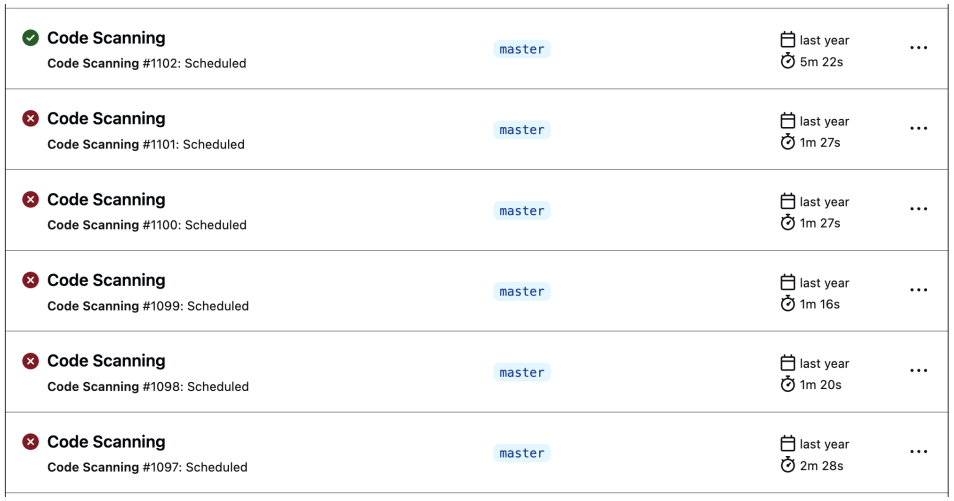

Fig. 1. An example of Workflow reruns.

a series of *steps*, which are the smallest units of work in a workflow. Steps can be defined by specifying the shell commands that should be executed or by delegating tasks to *actions* defined in the same repository, a public repository, or a published Docker container image. These *actions* are plug-ins that perform frequently repeated tasks [36]. The syntax of the workflow file can be seen from the GitHub documentation [37].

2.1 Workflow Reruns

If developers suspect that a GHA workflow run failure is unrelated to changes in the repository, they can request a rerun of the failed workflow. This can be done manually by re-triggering the workflow or automatically through trigger events (e.g., "schedule") [21]. Figure 1 provides an example of observed workflow reruns [23]. Over a period of six days, this workflow was executed six times without any updates to its original repository. As shown in the figure, the first five runs all led to failures, while the last one was successful. By inspecting the repository, we found that the `build` step in the workflow adopts an *action* located in another public repository. From one moment, the `npm` dependencies used in this *action* were outdated. Consequently, the workflow execution kept failing until the dependency issue in the adopted *action* was fixed. If the developer was aware of the issue earlier, the four subsequent failed reruns could have been avoided.

2.2 Job Reruns

Since the release of the job-rerun function on GitHub from 2022, developers can now rerun the failed job(s) instead of the whole workflow by clicking buttons on the GHA web UI. Reruns are particularly helpful when the workflow execution has non-deterministic (flaky) outcomes. Figure 2 presents an example of job reruns [22]. The drop-down list on the right displays the history of attempts for the current workflow execution. In this example, the job "*Test for tag transmission rocketmq*" failed in the first attempt. After rerunning this single job for extra three times, there was finally a successful outcome. By inspecting the workflow execution logs, we found the issue was caused by a flaky test related to `rocketmq`.

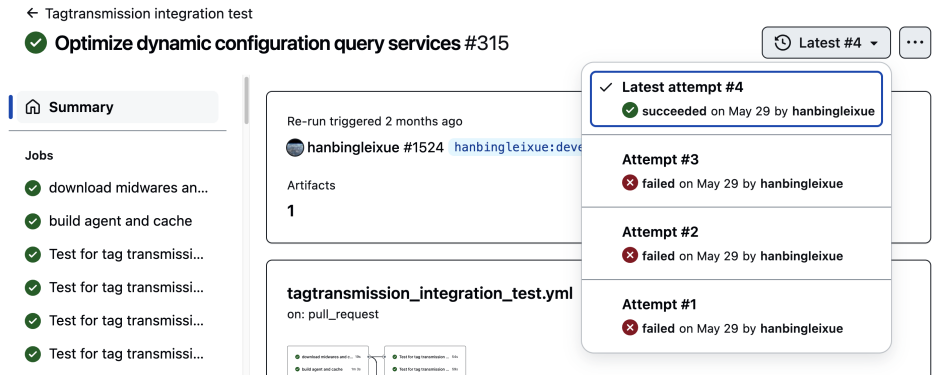

Fig. 2. An example of Job reruns.

2.3 Waste generated by Workflow & Job Reruns

While rerunning workflows or jobs can save developers' effort in investigating the reasons behind GHA workflow execution failures, blind reruns waste both time and computational resources if the failures are indeed caused by defects within the repository, as the workflow execution outcome is likely to remain unchanged. According to Sedano et al. [68], such unnecessary reruns can be considered a form of software waste. Moreover, in the case of private repositories, developers will be billed for any GHA usage that exceeds the allowed storage or minutes [4, 12].

3 STUDY DESIGN

The *goal* of this study is to understand how developers interact with GHA workflow failures and gain insights on potential means to prevent unnecessary workflow reruns for saving both time and computational resources. The *context* of this study consists of 9,202 distinct GHA workflow files collected from 3,320 GitHub repositories.

3.1 Research Questions

We aim at answering the following research questions (RQs):

- **RQ$_1$**: *How often do developers rerun workflows & jobs to handle GHA execution failures and what are the associated impacts?* This RQ aims to provide a comprehensive picture on how developers respond to the workflow execution failures. We seek to visualize the life cycle of a workflow execution and provide insights into the efficiency and effectiveness of workflow & job reruns during the project development process.
- **RQ$_2$**: *What are the reasons behind the flakiness of GHA workflow executions?* Given that the workflow executions sometimes generate inconsistent results (failing or succeeding unpredictably), this RQ aims to understand the factors behind the flaky GHA executions. Our goal is to better understand the circumstances under which reruns could effectively resolve issues, identify scenarios where alternative solutions might be more efficient, and understand ways to mitigate the flaky behaviors.
- **RQ$_3$**: *To what extent can we predict the outcome of a GHA workflow execution?* As we aim to avoid unnecessary reruns to save time and computational resources, this RQ explores the effectiveness of machine learning models for predicting the outcome of GHA workflow executions.

3.2 Data Preparation

To answer the proposed RQs, we need to collect a large dataset of GitHub projects employing GHA. To do so, we used SEART GitHub Search Engine [8] to select relevant GitHub repositories. In order to address common threats associated with mining software repositories [50], we excluded toy projects or projects with minimal development activities. More specifically, we adopted the following constraints:

- **Programming language**. In this study, we focus on Java projects because it is one of the most popular programming languages, used by millions of developers globally [39, 40]. Due to its scalability and stability, Java is also extensively used in enterprise environments [49], where automated CI/CD pipelines, including those powered by GitHub Actions, are essential. Prior CI-related studies have similarly restricted their scope to a single programming language [62].
- **Development activity.** We select projects which were created before 2023 and were still active (last commit within 1 month) at the time of data collection to ensure that there is enough development history to study the GHA usage. Besides, we excluded forked projects to prevent redundant data.
- **Popularity.** To ensure the relevance and maturity of the analyzed repositories, we include only projects with at least 100 stars and 100 commits. Prior studies have adopted similar method to exclude inactive or toy projects [51, 55, 70].

The final list included 5,217 repositories, of which 3,320 contained GHA workflow files, forming our curated dataset. Table 1 summarizes key statistics of the 3,320 repositories with GHA workflows. As mentioned, all selected repositories have at least 100 stars, ensuring a baseline level of community interest. Moreover, the dataset also exhibits diversity in popularity and development activity, with wide ranges in stars, forks, contributors, pull requests, and issues. This variety supports a representative analysis across projects of different scales and maturity.

For every working branch of these 3,320 repositories, we exploited the GitHub Rest API [32] to retrieve all WFs with their corresponding execution histories and associated commit logs throughout the whole year of 2023. Commit logs were used to identify repository changes between workflow executions, helping to determine whether a workflow execution is indeed a rerun (*i.e.,* re-executing the workflow without changing the repository content). In total, we collected 9,202 distinct WFs, and 5,576 out of them (60.6%) have at least one failed run.

Table 1. Overview of the 3,320 repositories containing GHA workflow files

Features	Mean	Q1	Median	Q3	Max	Min
# Stars	1,525	168.75	347	974.25	136,954	100
# Forks	465	53	114	303	44,765	1
# Contributors	46	10	24	51	483	1
# Pull Request	650	56	179.5	554	24,655	0
# Issues	540	42	137.5	437.5	95,431	0

In this study, we are interested in the scenarios of rerunning both the whole workflows and the selected failed jobs. As GitHub API does not provide an intuitive way to retrieve these types of reruns for GHA workflows, here we explain how we determine whether a GHA re-execution is a "*workflow rerun*" or a "*job rerun*":

Workflow rerun: *A workflow rerun occurs when a WF in a repository is executed again after its previous run has completed, without any updates to the repository.* Workflow rerun can be triggered either manually or automatically, depending on the trigger event. Based on observations on our collected dataset, we identified two scenarios where a new workflow execution can be identified as a rerun:

a) Two consecutive executions of a single WF have identical SHA values in the data retrieved with the GitHub API. This suggests that the workflow was triggered by an event other than a commit (*e.g.,* manually, issues, by schedule, etc.).
b) Two consecutive executions of a single WF have different SHA values, but no changes have been made to any file in the repository, as indicated in the commit logs. This suggests an empty commit was made solely to trigger the workflow.

Job rerun: *A job rerun occurs when, after the completion of a workflow run in a repository, one or more jobs within the same workflow are executed again without any updates to the repository.* Job rerun can only be triggered manually.

In practice, if a job has been rerun by developers, the `run_attempt` parameter of the job in the API data will have a value ≥ 2. This number begins with 1 and increments with each job rerun. We use this parameter to identify job reruns and distinguish them from workflow reruns.

During data collection, we observed that WFs were often rerun for multiple times until a desired outcome was achieved. To better analyze all the reruns following a failed workflow run, we introduce the concept of the **rerun set**: *all reruns following the initial failed workflow run.* We used Python scripts to automatically retrieve rerun sets for both workflow and job reruns from the previously collected execution history of WFs.

3.3 Replication Package

To support replication studies and future extensions, our replication package, including the scripts used and the dataset, is available online at https://anonymous.4open.science/r/Reruns-of-GHA-workflow-F0B1.

4 WORKFLOW & JOB RERUNS

To answer RQ1, we conducted an empirical study to investigate how developers deal with GHA workflow failures and the corresponding implications. More specifically, this RQ can be split into 3 sub-questions:

- **RQ**$_{1.1}$: How prevalent are workflow & job rerun in GHA?
- **RQ**$_{1.2}$: To what extent do workflow & job reruns change the workflow execution outcome?
- **RQ**$_{1.3}$: What are the software waste associated with workflow & job reruns?

4.1 Workflow & Job Rerun Prevalence (RQ$_{1.1}$)

Figure 3 depicts the life cycle of GHA workflow executions. In the data we collected, there are five types of different execution outcomes: `success`, `failure`, `cancelled`, `skipped`, and `startup_failure`, the definition of each outcome are presented as follows:

- `success`: The workflow completed all steps without errors.
- `failure`: The workflow ran but encountered an error and failed.
- `cancelled`: The workflow was manually or automatically stopped before finishing.
- `skipped`: There are steps or jobs within the workflow that were not run due to conditions (*e.g.,* `if:` expressions) evaluating to false.
- `startup_failure`: The workflow could not start at all due to setup or configuration issues.

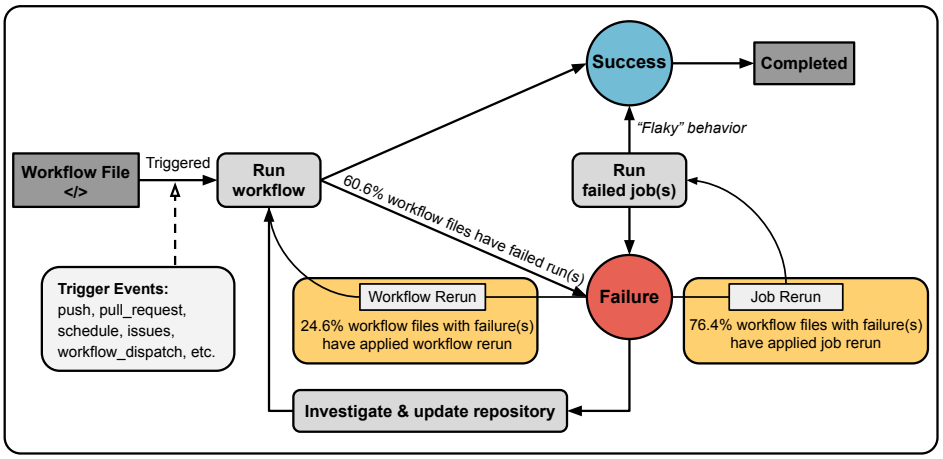

Fig. 3. Life cycle of a Workflow Run.

Table 2. Number of workflows with reruns

Rerun Type	Number of WFs	Percentages
Only workflow	881	15.8%
Only job	3,770	67.6%
Both types	491	8.8%
No reruns	434	7.8%

Among these outcome types, `success` and `failure` are the most common and typical, together accounting for 95.3% and 97.5% of the cases for workflow reruns and job reruns, respectively. Therefore, in this study, we focus only on these two types.

Of the 9,202 WFs collected, 60.6% (5,576) of them contain execution failures, indicating that GHA workflow failures are common in practice. When dealing with execution failures, developers could investigate the root cause of the failures and update the code repository, or simply rerun the whole workflow or failed jobs. To further understand how common it is to rerun workflows and jobs, we list the rerun frequencies in Table 2.

(Observation 1) Workflow rerun and job rerun occur in 24.6% and 76.4% of WFs, respectively. Among the 5,576 WFs containing at least one failed execution, 15.8% (881) only reran the whole workflows subsequent to the execution failures. As for job reruns, the number is much higher: a substantial proportion of WFs (67.4%, 3,770) only reran jobs. Additionally, 8.8% (491) of WFs have applied both workflow and job reruns, while only 7.8% (434) of WFs did not employ any form of reruns. Given that the job rerun function was only introduced recently in 2022, this observation reveals that developers prefer to rerun failed jobs to rerunning the whole workflow. Besides, the fact that 92.2% WFs contain workflow or job reruns indicate how common this practice is and suggest huge potential resource waste if reruns could not lead to desired outcomes.

(Observation 2) 54.8% of the workflow reruns are triggered fully automatically. As previously discussed, unlike job reruns, workflow reruns can be triggered either manually or automatically. We define a fully automatic workflow rerun set as one in which all reruns are

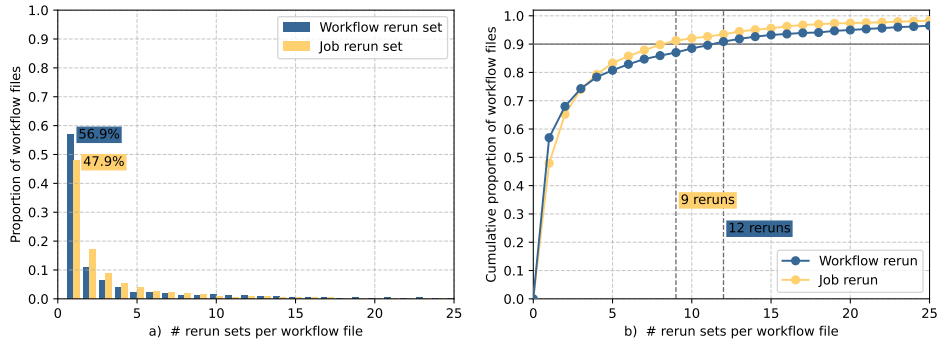

Fig. 4. Distribution of # rerun sets in WFs.

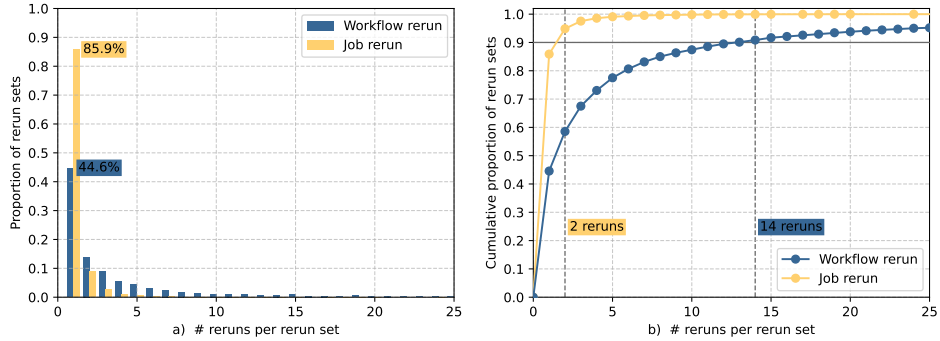

Fig. 5. Distribution of # reruns in rerun sets.

triggered by the "schedule" event. These reruns occur without human intervention, making them more prone to unnoticed failures and resource waste. We are particularly interested in the proportion of such fully automatic rerun sets. Our results show that more than half of the rerun sets (3,453 out of 6,302; 54.8%) fall into this category. This finding suggests that a significant portion of workflow reruns happen automatically, potentially leading to considerable time and energy waste. We believe that implementing smarter rerun mechanisms, especially for workflows scheduled at fixed intervals, could help reduce unnecessary executions.

Since it is clear that most WFs have gone through workflow and/or job reruns, we further looked into how often these reruns occur in each WF. Figure 4 shows the relative frequencies (Figure 4.a) and cumulative frequencies (Figure 4.b) of the number of rerun sets in each WF.

(Observation 3) On average, WFs with workflow reruns tend to have slightly more rerun sets compared to those with job reruns (4.6 vs. 4). That is, over the whole history of a WF, on average 4.6/4 execution failures have been followed by workflow/job reruns, respectively. The data also reveal that around half of the WFs with reruns (56.9% for workflow reruns and 47.9% for job reruns) contain only one rerun set during its history. Cumulatively, over 90% of WFs consist of ≤ 9 workflow rerun sets or ≤ 12 job rerun sets.

Figure 5 illustrates the distribution of the number of runs in each rerun set for both workflow and job reruns.

Table 3. Whether outcomes changed after reruns

Rerun Type	Execution Outcome	Count	Percent.
Workflow	Failure → Success (changed)	2,137	33.9%
	Failure → Failure (unchanged)	4,165	66.1%
Job	Failure → Success (changed)	12,537	73.6%
	Failure → Failure (unchanged)	4,497	26.4%

(Observation 4) On average, a workflow rerun set contains 7.9 workflow reruns, a job rerun set contains 1.3 job reruns. The data disclose that when developers rerun whole workflows, in 44.6% of the cases the rerun only occurs once after the failure; while for job reruns, the ratio raises to 85.9%. Over 90% of the job rerun sets consist of ≤ 2 reruns, while the number reaches 14 for workflow rerun sets. The significant disparity suggests that the number of job reruns applied by the developers is much smaller than that of workflow reruns, indicating the potential greater effectiveness of job reruns.

We also looked into the branches of the repository when reruns occur.

(Observation 5) 60% workflow reruns and 53.9% job reruns happened in default branches, respectively. Out of the 1,372 WFs with workflow reruns, 59.8% (821) were located in the default branch, with the remaining 40.2% distributed across other branches. Regarding job reruns, the proportion stood at 54.2% (2,308 out of 4,261). The predominant occurrence of reruns in the default branch can be attributed to the fact that most workflows (71.3%) are located in default branches. Another factor which can attribute to this is the scheduled workflow executions. In many projects, a scheduled time is set up to automatically trigger a workflow. However, these scheduled workflows can only run on the default branch.

4.2 Workflow & Job Rerun Effectiveness ($RQ_{1.2}$)

To understand whether workflow and job reruns lead to outcome changes, we extracted the execution results of the last rerun in each rerun set. Table 3 illustrates the outcome changes resulted from the executed reruns.

(Observation 6) 33.9% of the workflow rerun sets and 73.6% of the job rerun sets led to outcome changes. The observation was made on 6,302 workflow rerun sets and 17,034 job rerun sets extracted from our dataset. The data suggest that only 33.9% (2,137 out of 6,302) of the failed workflow executions turned out successful when developers rerun the whole workflows. When rerun jobs, the outcomes were significantly improved, namely that 73.6% (12,537 out of 17,034) of job reruns was successful in the end. We denoted the rerun sets with the outcome altered as **successful rerun sets**.

(Observation 7) On average, a successful workflow rerun set contains 11.6 workflow reruns, whereas a successful job rerun set contains merely 1.3 job reruns. As depicted in Figure 6, 36.7% of successful workflow rerun sets contain only one rerun, while the ratio is 86.6% for successful job rerun sets. Cumulatively, over 90% of successful job rerun sets have ≤ 2 reruns, whereas only half of workflow rerun sets adhere to this criterion. The difference between the sizes of successful workflow and job rerun sets is notably larger compared to the overall rerun sets, which might further reaffirm the effectiveness of job reruns.

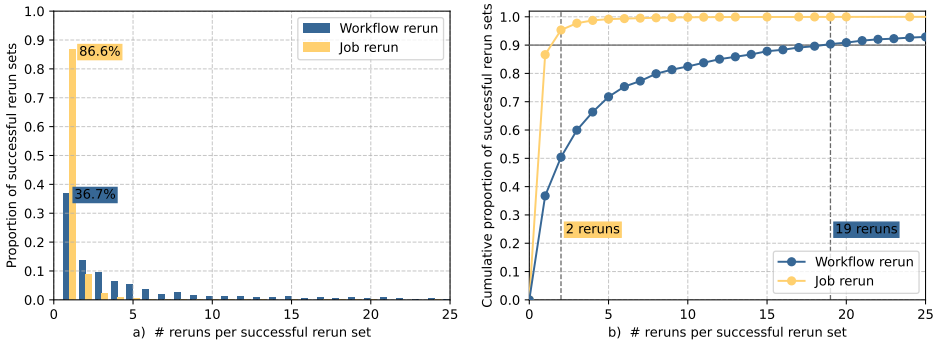

Fig. 6. Distribution of # reruns in successful rerun sets.

Table 4. Waiting time generated by reruns

Rerun Type	Outcome	Average Time	Total Time
Workflow	Changed	2 hours	0.5 years
	Unchanged	7.5 hours	3.2 years
	All	5.3 hours	3.7 years

4.3 Overheads generated by Workflow & Job Reruns ($RQ_{1.3}$)

Following the prior work[68], we quantified the waste generated by reruns in terms of waiting time and computational time. The *waiting time* represents the duration between triggering the rerun and the end of workflow execution, which could be obtained via the GitHub API using the `run_duration_ms` key. The *computational time* is the sum of execution duration of all jobs in a workflow. The execution duration of a single job can be calculated based on job start time and job end time (retrieved from `started_at` and `completed_at` properties of each job using GitHub API). The waiting time and the computational time differ because 1) the former also includes the time spent in waiting for a GitHub Actions Runner to pick up the tasks due to limited computing resources; 2) individual jobs within a workflow can be run in parallel. For workflow reruns, we considered both waiting time and computational time. While for job rerun, we only calculated the computational time of the jobs as the waiting time of the previous execution attempts are untraceable due to GitHub platform limitations.

(Observation 8) Around 3.7 years were spent waiting for workflow reruns. Table 4 shows the waiting time of workflow reruns throughout 2023. While developers spent around 3.7 years on waiting for the outcome of workflow reruns, around 3.2 years was spent on rerun sets that failed to change the outcomes. Only 0.5 years were dedicated to successful rerun sets. On average, the waiting time for a workflow rerun set was 5.3 hours, while the duration was reduced to 2 hours for successful rerun sets. The rerun sets failing to alter outcomes require significantly more average waiting time of 7.5 hours.

(Observation 9) Around 2.7 years and 1.8 years of computational time were spent on workflow reruns and job reruns, respectively. As shown in Table 5, the total computational time spent on workflow reruns tallied up to 2.7 years, of which only 1.1 years spent for rerun sets resulting in outcome changes. The average computational time of a single workflow rerun set was 3.9 hours. Interestingly, the difference in computational time between rerun sets with changed and

Table 5. Computational time generated by reruns

Rerun Type	Outcome	Average Time	Total Time
Workflow	Changed	4.4 hours	1.1 years
	Unchanged	3.6 hours	1.6 years
	All	3.9 hours	2.7 years
Job	Changed	0.8 hours	1 year
	Unchanged	1.3 hours	0.8 years
	All	0.9 hours	1.8 years

unchanged outcomes (0.9 hours) is less pronounced compared to their difference in waiting time (5 hours). In terms of job reruns, the total computational time amounted to 1.8 years, with 1 year dedicated to rerun sets with changed outcomes. Despite having nearly three times as many job rerun sets in our dataset compared to workflow rerun sets (17,034 vs. 6,300), the total computational time of job reruns is much smaller than that of workflow reruns (1.8 years vs. 2.7 years). On average, the execution time for a job rerun set was 0.9 hours, while the time spent on successful rerun sets was 0.5 hours less.

By comparing the time wasted on workflow and job reruns, we can clearly see that that job reruns normally are more effective and generate less software wastes. However, the wastes are still significant, indicating the importance of reducing them.

4.4 Discussion

4.4.1 Waiting Time in Unchanged Workflow Reruns. Table 4 shows that workflow reruns with unchanged outcomes exhibit substantially longer waiting times than those with changed outcomes (average waiting time of 7.5 hours versus 2 hours). Further investigation revealed that this difference is primarily caused by a subset of workflows that experience exceptionally long execution durations. Such cases typically occur when persistent configuration or environment issues remain unresolved across repeated executions. For example, the workflow `example-sanity-check.yml` [19] from repository `deeplearning4j/deeplearning4j-examples` failed continuously because the required runner `ubuntu-18.04` had been officially deprecated[1]. Despite being scheduled to trigger daily, each run waited more than one day before being automatically terminated by GitHub. This single issue alone accounted for 757 days of wasted waiting time in our dataset. In contrast, other unchanged rerun sets showed an average waiting time of only 2 h 41 m 41 s. Regular maintenance, such as updating runner configurations, could have prevented these prolonged reruns.

4.4.2 CI or CD. Previous analyses revealed that waste in GitHub Actions (GHA) workflows is partly driven by triggering mechanisms. To further investigate this, we examined whether different workflow types contribute differently to the waste. Specifically, we compared Continuous Integration (CI) and Continuous Deployment (CD) workflows. Our findings show that CI workflows account for approximately 79% (759.55 of 961.35 days) of the total computational time wasted due to workflow reruns, compared to 11.5% (110.78 days) from CD workflows. The remaining 9.5% originates from workflows not clearly classified as either CI or CD. A similar pattern is observed for job reruns, with CI workflows responsible for 83% (316.62 of 382.94 days) of the waste, while CD workflows contribute just 2.9% (11.17 days). These results suggest that CI workflows are the

[1]https://github.com/actions/runner-images/issues/6002

primary source of inefficiency, likely due to their frequent execution in response to events such as pull requests and code commits.

4.4.3 Caching and Workflow Waste. Although this study focuses on workflow reruns and their contribution to waste, it is also worth considering the role of optimization techniques such as caching. Caching is commonly used in GitHub Actions workflows to reduce redundant computations or downloads and improve efficiency. However, its impact on overall resource waste is not always clear. While caching can speed up execution, it does not necessarily reduce wasted resources, particularly when other inefficiencies, such as flaky tests or unnecessary reruns, are present [1]. This highlights the complexity of evaluating actual resource savings and suggests that improving test stability and rerun policies may be just as important as implementing caching. Future work could further explore how caching interacts with these factors to better understand its effectiveness in reducing workflow waste.

Summary RQ_1: How often do developers rerun workflows & jobs to handle GHA execution failures and what are the associated impacts?

Workflow reruns occur in 24.6% of workflows, while job reruns are more common, occurring in 76.4% of cases. More than half of workflow reruns are triggered fully automatically, and workflows with reruns tend to have slightly more rerun sets than those with job reruns, averaging 4.6 compared to 4. Each workflow rerun set contains an average of 7.9 reruns, whereas job rerun sets contain about 1.3 reruns. The majority of reruns take place on default branches, with 60% for workflow reruns and 53.9% for job reruns. Outcome changes occur in 33.9% of workflow rerun sets and 73.6% of job rerun sets. Successful rerun sets are larger for workflows, averaging 11.6 reruns, compared to just 1.3 for jobs. In total, approximately 3.7 years of waiting time were spent on workflow reruns, 2.7 and 1.8 years of computational time were spent on workflow and job reruns, respectively. These findings highlight that while job reruns are more frequently adopted, workflow reruns involve more extensive repeated executions and consume significantly more resources, suggesting a potential area for optimization in workflow management to improve efficiency.

5 FLAKINESS OF WORKFLOW EXECUTIONS

Section 4 shows that the workflow run outcomes transitioned from failure to success in 33.9% of the workflow rerun sets and 73.6% of the job rerun sets, although no changes were made to the code repository. The altered outcomes imply the existence of flakiness in workflow runs. In this section, we conduct manual analysis to understand the reasons behind the flaky behavior of workflow executions. Similar to security misconfigurations that persist across workflow revisions [47], our analysis reveals that many causes of workflow execution flakiness arise from external dependencies, environment instability, or configuration issues rather than from source code changes. More specifically, we categorize the root causes of flakiness in both workflow and job reruns, count their frequency of occurrences, and provide insights on how to eliminate them.

5.1 Study Design

5.1.1 Data Collection. To gain a more comprehensive understanding of workflow execution flakiness, we extracted all sequences of workflow executions (denoted by "**flakiness set**") in which initially failed runs turned out successful in the end from the previously collected rerun sets. As an example, if a rerun set has a series of outcomes *[failure, failure, success, failure, success]*, it is divided into two separate flakiness sets: *[failure, failure, success]* and *[failure, success]*. This resulted in 2,344

distinct workflow flakiness sets and 5,557 job flakiness sets. These flakiness sets were sorted by the number of failures in each set. As the flakiness sets with a larger number of failed reruns are considered to generate more wastes, we prioritized the analysis for these sets.

5.1.2 Data Analysis. We manually analyzed the reasons behind the flakiness by inspecting the WFs, the relevant workflow execution logs, the associated commits, and the dependencies used in the workflow. One author with 8 years' software development experience and 3 years' experience with CI/CD conducted the initial analysis of each flakiness case. A second author, also highly experienced in this domain (15 years' software development experience and 10 years' CI/CD experience), independently reviewed and verified the identified root causes. In instances of disagreement, both authors engaged in a thorough re-analysis and discussion of the specific case to reach a well-reasoned consensus, ensuring the reliability and consistency of the classification. Manually identifying root cause is time-consuming as several data sources need to be carefully checked. Inspecting each flakiness set takes up to 3 hours.

Given the large number of flakiness sets, it is impractical to manually analyze all of them. Therefore, we adopted a saturation strategy inspired by Guest et al [43]. Specifically, we set an initial sample of 30 flakiness sets which contain the most failed reruns, with a stopping criterion of 10 following sets, and a 100% threshold.

If the causes identified in the initial 30 flakiness sets included all causes extracted from these 40 sets, we determined that the saturation had been reached. If the saturation criterion was not satisfied, we analyzed 10 more sets and compared the causes identified from the 50 sets with those identified in previous 40 sets.

Eventually, for both workflow and job reruns, we investigated a total of 50 flakiness sets, reaching saturation after inspecting 40 sets. Due to the limited size of the initially investigated flakiness sets, we have doubled the number of manually analyzed sets to ensure a more robust conclusion. In total, we investigated 100 flakiness sets for both workflow and job reruns. To clearly present our results and highlight the insights, the following notations are used:

- ✱: provides an example of workflows which fit into the current category of root causes of workflow run flakiness.
- ❖: indicates potential solutions or directions to address some of the flakiness.

Note that the potential solutions we present are based on both the recurring patterns we observed during our analysis and our prior experience in diagnosing and mitigating workflow-related issues. They are intended to offer practical guidance for addressing specific causes of workflow run flakiness and to inform future improvements in workflow reliability.

5.2 Flakiness in Workflow Reruns

Figure 7 presents the distribution of seven types of root causes of workflow flakiness identified within the top 100 distinct workflow flakiness sets, including test flakiness, external dependency issue, workflow dependency issue, branch discrepancy, external service unusable, dynamic external inputs, GitHub API rate limitation and mishandling edge cases. For 10% of the cases, we were unable to identify the root causes due to deleted logs/execution records or lack of information.

5.2.1 Test Flakiness. Test flakiness accounts for 32% of the investigated cases. In this scenario, the workflow typically automates the build process of the project within the repository. If the build consists of flaky tests, the corresponding workflow will also have non-deterministic outcome.

✱ For instance, workflow `nightly.yml` [24] of the repository `firebase/firebase-admin-java` automates the build process using Maven. However, sometimes flaky internal tests returned errors, causing build failures and consequently workflow execution failures.

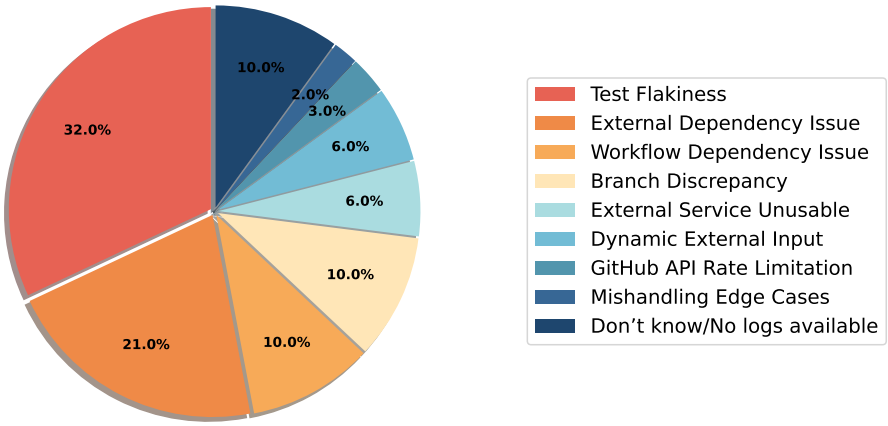

Fig. 7. Distribution of causes of flakiness - Workflow reruns

✱ Another example can be found in `build-and-test-scheduled-3.2.yml` [14] of the repository `apache/dubbo`, where developers schedule the use of the matrix function to test and build the project with various versions of JDK. Occasionally, test errors occur when executing unit tests on certain system-version combinations, resulting in workflow run failures.

Test flakiness is an active research topic in software engineering community, which needs significant future effort to be effectively addressed. The test flakiness can arise from various factors. The primary cause we observed was the time out issues. Other types of test flakiness include connection errors, file downloading issues, etc.

❖ *When automating builds in workflows, developers may reduce or skip flaky tests, especially in scheduled workflows.*

5.2.2 External Dependency Issue. The second most common cause is external dependency issue, it represents 21% of the cases. GitHub's flexibility allows integration with numerous external dependencies (*e.g., actions*, *dockers*, etc.), which can cause failures if they contain issues.

✱ Workflow `graalvm-dev.yml` [28] of the `micronaut-projects/micronaut-spring` repository utilized several custom *actions* [33], two of which (`build-matrix` and `pre-build`) had configuration issues. The workflow stopped failing once these issues were resolved [35].

❖ *Developers may regularly upload the version of external dependencies used in workflows to avoid execution failures caused by dependency bugs.*

5.2.3 Workflow Dependency Issue. 10% of the flakiness sets fall in this category, where a workflow is triggered by the completion of another workflow's execution through the event `workflow_run`, typically using the output of the dependent workflow. Sometimes unexpected outcomes could lead to flakiness in the current workflow.

✱ In the repository `elastic/apm-agent-java`, the execution of `test-reporter.yml` [20] is triggered when another workflow `main.yml` is completed, regardless of whether it succeeds, fails, or is canceled. Its functionality is to generate test reports based on the test results returned by `main.yml`. However, when `main.yml` is canceled and no test results are generated, `test-reporter.yml` will also fail.

❖ *If a workflow depends on the completion of another workflow, developers can use conditions to control job execution (*e.g.*, skip the workflow if the dependent workflow fails or returns unexpected values) to avoid unnecessary executions.*

5.2.4 Branch Discrepancy. This category affects 10% of cases. GHA workflows located in one branch but designed to operate on another branch can become unreliable, as changes in the target branch can impact workflow run outcomes, even though no modifications was made to the original branch.

✱ For example, workflow `nightly.yml` [2] defined in the default `main` branch of the repository `mediathekview/MediathekView` [27] aimed to automate nightly build and release processes for the `develop` branch. The Java version specified in the workflow was originally consistent between two branches. However, conflicts arose when configurations in the `develop` branch was altered, leading to workflow failures.

✱ In the workflow `fetch-dev-repo.yml` [17] of the repository `corretto/corretto-8`, developers scheduled an automated branch merge from the "upstream-dev" branch to the "nightly" branch. Whenever there were merge conflicts, the merge process could not be performed properly, resulting in errors, ultimately caused workflow run failures.

According to GitHub's policy governing workflow runs, GitHub restricts the triggering of some events (*e.g.,* `push`, `pull_request`, `schedule`, etc.) to the workflows located solely on the default branch to ensure stability for critical operations. However, this can lead to workflow flakiness, as developers might not always synchronize the settings of different branches.

❖ *Tools could be developed to detect misalignment of settings in different branches, and alert developers timely or automatically update the workflow in the default branch.*

5.2.5 External Service Unusable. 6% of the flakiness sets fall into this category, where the workflow attempts to interact with external services (*e.g.,* remote servers) but fails.

✱ Workflow `links-check.yml` [13] in `aklivity/zilla` repository encountered issues accessing a remote server during execution, resulting in errors with return code 503 (Service Unavailable).

❖ *Developers should monitor external service availability and consider separating remote-dependent steps from the main CI pipeline to avoid impacting overall reliability.*

5.2.6 Dynamic External Inputs. This category represents 6% of cases. In GHA, workflows can be designed to receive inputs from practitioners. However, those inputs can lead to flakiness if they do not meet specific criteria.

✱ Workflow `upgrade-bridge.yml` [30] in the repository `pulumi/pulumi-gcp` has trigger event `repository_dispatch`, which allows users to pass parameters for executing workflows. It fails when those inputs contains unrecognized field names.

❖ *Conditions could be applied to verify user input validity in workflows to avoid unnecessary reruns.*

5.2.7 GitHub API Rate Limitation. Another 3% of the flakiness sets fall in this category. Some workflows contain scripts or *actions* that requires using GitHub Rest API, which has a rate limit of 5,000 accesses per hour. Once the limit has been reached, GitHub will block the execution process of the original workflow.

✱ In the repository `awslabs/aws-advanced-jdbc-wrapper`, `remove-old-artifacts.yml` [15] workflow automatically removes artifacts older than one week from the repository using GitHub API. Occasionally, it exceeds the access limit, causing workflow execution failures.

❖ *Developers can monitor the GitHub API quota constantly and schedule the workflow executions accordingly.*

[2]This workflow was later deleted by the author on May, 2024.

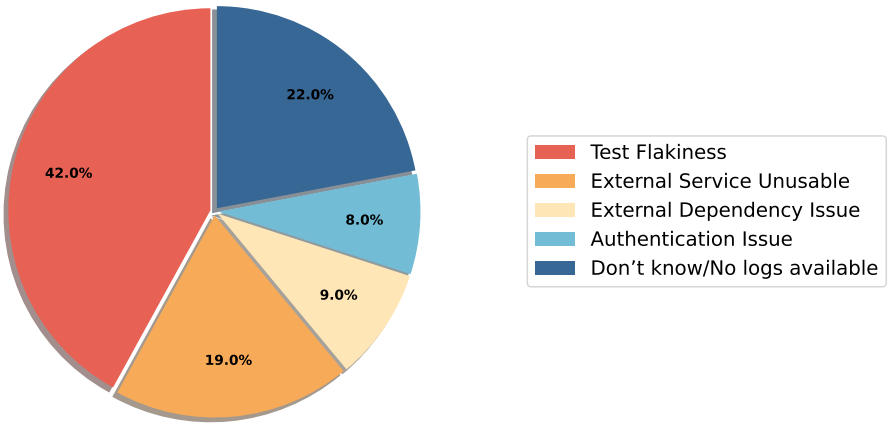

Fig. 8. Distribution of causes of flakiness - Job reruns

5.2.8 Mishandling Edge Cases. This category represents 2% of flakiness sets. Some workflows may not handle edge cases well. For example, when an issue-triggered workflow contains a job to read issue description and the description is left empty. The workflow execution might fail if this case is not considered, although this does not really impact the project itself.

✱ For instance, the workflow `maven.yml` [29] in the repository `niumoo/bing-wallpaper` fetches new wallpapers to commit to the repository, however, when there is nothing to commit, GitHub commit command will return an error, leading to workflow failures.

❖ *Developers should better handle edge cases or sad paths in their workflows.*

5.3 Flakiness in Job Reruns

Figure 8 demonstrates the five causes of flakiness identified within the top 100 job flakiness sets, including test flakiness, external service unusable, authentication issue, external dependency issue. Root causes can not be identified for 22% of the flakiness sets due to lack of information.

5.3.1 Test Flakiness. Similar to the flakiness in workflow reruns, the predominant cause of flakiness in job reruns is also test flakiness, accounting for 42% of all cases.

✱ For instance, in the workflow `gradle.yml` [38] of repository `vividus-framework/vividus`, developers automated the build process of the project using Gradle. Occasionally, internal integration test errors occurred due to unstable connection, leading to build failures and then workflow execution failures.

❖ *Similar to flakiness in workflow reruns, developers should manage the flaky test to a minimum occurrence rate.*

5.3.2 External Service Unusable. 19% of the flakiness sets fall into this category, where the workflow attempts to use external services such as AWS but fails.

✱ Workflow `llm_integration_p4d.yml` [18] in the repository `deepjavalibrary/djl-serving` throw an exception `ReservationCapacityExceeded` during its execution, indicating that the AWS reservation capacity was exceeded, meaning there were not enough available resources to fulfill the request.

❖ *Developers may reduce workflow disruptions by setting up uptime checks for external services and moving dependent tasks into isolated jobs outside the main CI flow.*

5.3.3 External Dependency Issue. In job reruns, 9% of flakiness cases fall into the category where workflow runs fail due to issues generated by external dependencies.

✱ Workflow `javacpp.yml` [16] of the repository `bytedeco/javacpp` used an external action that contained a bug, causing workflow failures. This was solved after the bug was fixed.

❖ *Same as in workflow flakiness sets, external dependencies should be verified regularly to avoid unnecessary reruns.*

5.3.4 Authentication Issues. In 8% of the cases, the workflow execution is stopped due to failed authentication.

✱ `snapshot-e430-eclipse-distro-build.yml` [34] of the repository `spring-projects/sts4` is designed to build a snapshot with inherited secrets which consist of a token. Once the token has expired, a fatal error occurs that stops the process, leading to workflow failures.

❖ *Developers should verify authentications regularly before running workflows in order to avoid unnecessary reruns.*

Summary RQ_2: What are the reasons behind the flakiness of GHA workflow executions?

We identified various root causes of workflow execution flakiness by manually inspecting 100 distinct workflow rerun sets and 100 distinct job rerun sets, each selected based on the highest frequency of reruns. At the workflow level, the causes include test flakiness (32%), external dependency issue (27%), workflow dependency issue (11%), branch discrepancy (9%), dynamic external inputs (6%), GitHub API rate limitation (3%) and mishandling edge cases (2%). At the job level, flakiness is attributed to test flakiness (42%), external service unusable (17%), authentication issue (10%), external dependency issue (9%). Our findings highlight that test flakiness and external factors—both environmental and service-related—are dominant contributors to workflow execution flakiness, underscoring the need for more robust test design and resilient workflow engineering practices.

6 ML BASED WORKFLOW RUN OUTCOME PREDICTION

Given the fact that a large percentage of reruns do not lead to meaningful results as presented in Section 4, we aim to investigate to what extend we could predict the workflow execution outcomes to avoid unnecessary reruns and save both time and computational resources.

6.1 Study Design

To understand whether a ML-based model can effectively predict the workflow execution outcomes, we split RQ_3 into:

- $RQ_{3.1}$: To what extent can ML-based models accurately predict the outcomes of workflow executions?
- $RQ_{3.2}$: Which features contribute most significantly to the effectiveness of the models?

6.1.1 Data Collection. We curated the dataset for training ML-based models on top of the data collected for the empirical study in Section 4. For workflow reruns, this dataset comprises 50,631 workflow runs from 6,302 workflow rerun sets across 1,372 WFs. Among these workflow runs, 23,462 resulted in "success" while 27,169 ended in "failure", yielding a success rate of 46.34%. As for job reruns, the dataset contains 18,597 workflow runs from 7,708 job rerun sets across 4,260 WFs,

Table 6. Examples of dataset for workflow and job reruns

WF run	idx	Historical		Complexity		Churn		Outcome
Run #1	1364	1	2	26	1	3	1	0
Run #2	1142	1	8	50	2	5	0	1
Run #3	1361	2	162	56	1	3	1	1
Run #4	84	2	13	66	2	3	1	0
Run #5	1084	1	107	33	1	3	1	1
...	...	...	...	...	...	...	...	...

with a success rate of 34.05%. Table 6 demonstrates the features included in the binary classification task and their respective origins. Inspired by Anderson *et al.* [3], similar to the regression tests analyzed in their study, each workflow run record contains three key categories of information: historical data, complexity data, and churn data, with details explained below.

Historical features: Containing execution records of the workflow, capturing outcomes and runtime details from previous runs to support analysis and prediction of future behavior. This data consists of the following two features:

a) Last Run Outcome: The outcome of the workflow in its most recent run. Set as 1 for "success", 2 for "failure", and 0 if it is the workflow's initial run.
b) Number of Reruns: Number of reruns the workflow has undergone up to the current execution, 0 if it is the workflow's initial run.

Complexity features: Representing the structural complexity within the workflow, which may impact its execution behavior. This category contains the following two features:

a) Number of Lines: Number of lines of codes in the WF.
b) Number of Jobs: Number of jobs in the WF.

Churn features: Reflecting workflow activity and repository integration context, which may influence its stability and execution patterns. This category also contains two features:

a) Trigger Event: The event that triggers the workflow execution. 19 distinct types are found in our dataset, represented by an integer from 0 to 18.
b) Branch Info: Whether the WF is located in the default branch of its repository, 1 for True, 2 for False.

These features are used to create classification models both individually and in combination to assess their effectiveness. Specifically, for each ML-based approach, three models are trained based solely on features from one category of data (historical, complexity or churn). Meanwhile, another model is constructed using all features from all categories. This approach results in a total of 4 different models. The *idx* column in Table 6 represents the id of the WF and is not used in training.

6.1.2 Data Splitting. To train and evaluate ML-based models, we split our dataset into train and test sets with a ratio of 4:1 on the level of distinct workflow runs. In total, for workflow reruns, there are 40,504 workflow run records in train set and 10,127 in test set; As for job reruns, there are 14,877 records in train set, 3,720 in test set.

6.1.3 Model Selection. Unlike prior learning-based approaches that assist developers during workflow authoring [46], the models evaluated in this study focus on execution-level features to predict workflow outcomes at run time. To this end, we investigate several machine learning models

commonly used for binary classification in software engineering. According to a recent survey on predictive modeling in software engineering, three widely adopted and representative classification algorithms in this domain are Naive Bayes (NB), Decision Trees (DT), and Support Vector Machines (SVM) [71]. In addition to these traditional models, we also included a basic neural network model named Multi-Layer Perceptron (MLP) to reflect the growing trend and popularity of neural networks in recent software engineering research.

Naive-Bayes models [56] are mathematical models that assume independence among attributes and calculate class probabilities based on attribute weights. Decision Tree models [69] are easily interpretable and illustrates feature importance through their tree structure. Support Vector Machine (SVM) models [6] are robust classification models that construct a decision boundary by maximizing the margin between classes, making them effective for high-dimensional data. MLP classifiers [57] are neural networks comprising an input layer, one or more hidden layers, and an output layer. Each neuron in a layer is connected to neurons in adjacent layers, with weights dictating the strength of these connections. MLP classifiers have demonstrated high accuracy in various classification tasks. By investigating these four models, we ensure a comprehensive comparison and validation of the ML-based models on workflow outcome prediction.

It is worth noting that in this study we intentionally excluded larger deep learning architectures such as CNNs, RNNs, Transformers, and large language models (LLMs). Although workflow files are textual YAML documents, our models operate on derived tabular features rather than raw text. In fact, our dataset combines information from multiple sources: only a subset of features (*e.g.,* number of lines, number of jobs, and trigger events) originate from workflow files, while others such as last run outcome, number of reruns, and branch information are collected from the GitHub API execution logs. This mixture of structured, categorical, and numerical attributes does not exhibit the spatial, sequential, or semantic regularities that deep neural architectures are designed to capture. Hence, employing such models would not provide meaningful representational advantages while introducing substantial computational overhead. In contrast, the compact models we selected are well suited to the limited feature space and preserve interpretability, which is essential for understanding workflow outcome predictions. Therefore, our choice of multiple models balances data suitability, architectural relevance, interpretability, and efficiency, ensuring both methodological soundness and scientific transparency.

6.1.4 Evaluation Metrics. In binary classification tasks, Precision, Recall, and F1-score are most commonly used metrics to assess the effectiveness of a model's predictions.

Precision measures the accuracy of positive predictions made by the model, it is calculated as the ratio of true positive predictions to the total number of positive predictions.

Recall measures the ability of the model to identify all relevant instances of the positive class, it is calculated as the ratio of true positive predictions to the total number of actual positive instances in the dataset.

F1-score is the harmonic mean of precision and recall, providing a balance between the two metrics. It ranges from 0 to 1, where a score closer to 1 indicates better performance, considering both precision and recall. F1-score is particularly useful when imbalance exists between the classes in the dataset.

In addition to these metrics, **confusion matrices** are also used to provide a detailed breakdown of prediction outcomes, showing the number of true positives, false positives, true negatives, and false negatives. This enables a more granular analysis of each model's strengths and weaknesses.

6.1.5 Experimental Implementation. We applied Scikit-learn [60] to implement the models:

- *Multi-Layer Perceptron (MLP):* We used *MLPClassifier* [66] with customized-hyper parameters. Details of the tuning process are provided below.
- *Naive-Bayes (NB):* We used *GaussianNB* [65] with default hyper-parameters.
- *Decision-Tree (DT):* We used *DecisionTreeClassifier* [63] with default hyper-parameters.
- *Support Vector Machine (SVM):* We used *SVC* [67] with default hyper-parameters.

Tuning hyper-parameter: For the MLP model, we performed a grid search to tune the hyper-parameters. Specifically, we experimented with different values for the number and size of hidden layers, batch sizes, and maximum iterations. The configuration that achieved the best performance on the validation set consisted of three hidden layers with sizes 50, 100, and 50 respectively, a batch size of 16, and a maximum of 300 training iterations. We enabled early stopping with a validation fraction of 0.2 to prevent overfitting during the tuning process. To ensure reproducibility, we set the random state to 1. All other hyper-parameters were kept at their default values.

Baselines: Two baselines are applied for comparison. All-Success and Straw-Man:

- *All-Success*: Adopting the concept of a dummy classifier [64], All-Success always predicts the outcome as a predefined class ("success" in our context), serving as a simple benchmark for evaluating more complicated models. For workflow reruns, given that our collected workflow runs have an success rate of 46.3%, All-Success yields a precision of 0.463 and a recall of 1, with a corresponding F1-score of 0.633; As for job reruns with success rate of 34.05%, this results in a precision of 0.341 and a recall of 1, with F1-score of 0.509.
- *Straw-Man*: Since many WFs are automatically triggered by scheduled events, their outcomes often mirror the results of the previous execution. To assess whether ML-based models can truly "learn" from diverse data types, we introduce the Straw-Man approach as the second baseline, which predicts the execution outcome of a WF solely based on its recent execution. If the last execution failed, the prediction would be negative, and vice versa. This model leverages historical data with simple if-else logic, which does not involve any training process.

6.2 Result Analysis

Table 7 and Table 8 demonstrate the overall performance for workflow execution outcome prediction of the four ML-based models and two baselines on both workflow reruns and job reruns, respectively. In each table, the best performances of the models using all features are in bold, while the best performances with individual features are underlined.

6.2.1 Model Evaluation. Overall, for workflow reruns, all four models achieve decent performance, with F1-scores ≥ 0.839. The MLP classifiers slightly outperform the other three traditional models in terms of F1-score and Precision, while the Decision Tree models achieve the highest Recall. Compared to the two baselines, we can see that all four ML-based models trained on all features have better performance in terms of F1-score.

The confusion matrices presented in Figure 9 provide deeper insight into the performance of the selected models. The four applied models consistently outperform the two baselines in both precision and recall. Among them, MLP, SVM and Decision Tree show the most balanced performance, while Naive Bayes exhibits more frequent misclassifications, particularly false negatives. In contrast, the Straw-Man baseline underperforms across all metrics, and the All-Success strategy fails to differentiate between classes, producing the weakest overall results. These results demonstrate the great potential of using ML models to effectively predict the outcome of the workflow execution.

When it comes to job reruns, the ML-based models still perform reasonably well, although the scores are lower than those for workflow runs. MLP and Decision Tree classifiers achieve F1-scores of 0.708 and 0.707, clearly outperforming both baselines. The Decision Tree has the highest Precision,

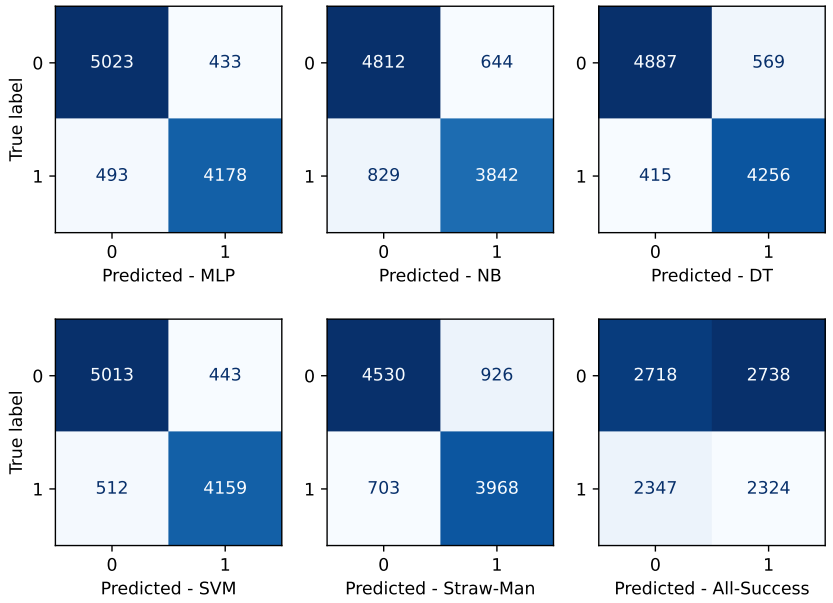

Fig. 9. Confusion matrix comparison across models - Workflow reruns

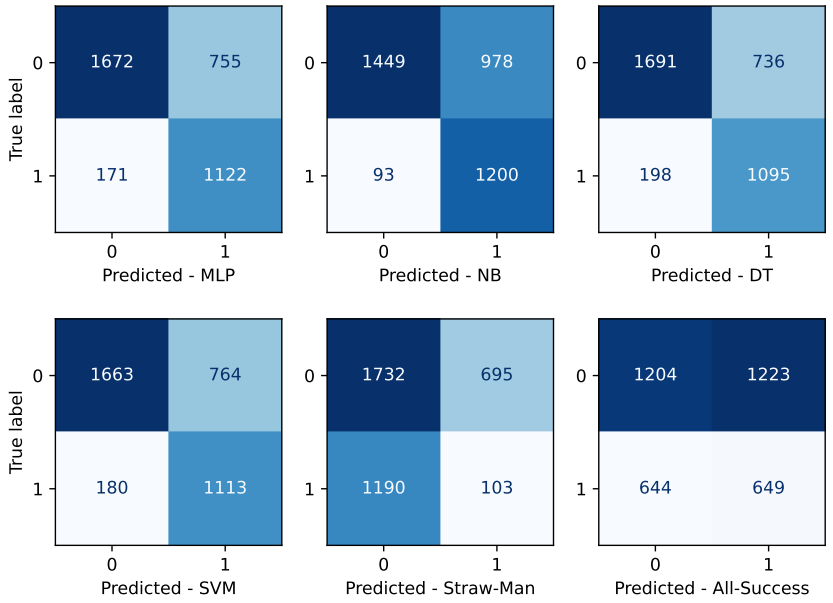

Fig. 10. Confusion matrix comparison across models - Job reruns

Table 7. Models' Performance in Predicting Workflow Outcomes @ Run level - Workflow rerun

Metric	Feature Cat.	MLP	NB	DT	SVM	All-Success	Straw-Man
F1-score	All	**0.9**	0.839	0.897	0.897	0.633	0.835
	History	0.887	0.87	0.887	0.887		
	Complexity	0.715	0.636	0.775	0.643		
	Churn	0.666	0.631	0.667	0.639		
Precision	All	**0.906**	0.856	0.882	0.904	0.463	0.821
	History	0.928	0.883	0.927	0.928		
	Complexity	0.608	0.482	0.71	0.478		
	Churn	0.501	0.462	0.502	0.47		
Recall	All	0.894	0.823	**0.911**	0.89	1	0.849
	History	0.85	0.857	0.849	0.85		
	Complexity	0.869	0.933	0.852	0.985		
	Churn	0.994	0.998	0.992	0.996		

Table 8. Models' Performance in Predicting Workflow Outcomes @ Run level - Job rerun

Metric	Feature Cat.	MLP	NB	DT	SVM	All-Success	Straw-Man
F1-score	All	**0.708**	0.698	0.707	0.702	0.509	0.1
	History	0.708	0.701	0.708	0.685		
	Complexity	0.516	0.516	0.516	0.516		
	Churn	0.516	0.516	0.516	0.516		
Precision	All	0.594	0.56	**0.598**	0.593	0.341	0.127
	History	0.59	0.568	0.594	0.624		
	Complexity	0.348	0.348	0.348	0.348		
	Churn	0.348	0.348	0.348	0.348		
Recall	All	0.883	**0.924**	0.864	0.861	1	0.083
	History	0.875	0.917	0.875	0.76		
	Complexity	1	1	0.998	1		
	Churn	1	1	0.998	1		

while Naive-Bayes reaches the best Recall at 0.924. History-based features contribute the most to performance, whereas complexity and churn features offer limited value.

Figure 10 presents the confusion matrices of the 6 models on job rerun outcome prediction. We can see that the four ML-based models again show clear advantages over the two baselines in classification performance. MLP, SVM and Decision Tree exhibit a strong balance between true positives and true negatives, whereas Naive Bayes performs slightly worse due to a higher number

of false positives. The Straw-Man and All-Success baselines demonstrate significantly weaker performance, with a high number of misclassifications in both classes. These results indicate that while predicting job-level outcomes is more challenging, ML models remain robust compared to naive predicting strategies, especially when using historical data.

Interestingly, we found that the Straw-Man baseline performs noticeably better on workflow rerun outcome prediction than on job rerun. This difference can be attributed to the nature of workflow reruns, which are predominantly automated and often triggered in response to the same conditions or outcomes, as discussed in Section 4, making the previous execution outcome a strong predictor for the next one. In contrast, job reruns are usually initiated manually, often after developers inspect logs or apply fixes, leading to less predictable behavior and weaker correlation with prior outcomes. As a result, the Straw-Man model, which naively relies on the previous outcome, fails to capture the more nuanced human-driven dynamics of job reruns, resulting in a sharp drop in performance.

6.2.2 Feature Contribution. As can be seen in Table 7, in terms of workflow reruns, for the best performing MLP, training on all features leads to better results than training on individual feature categories. When trained on only historical data, the four models can already achieve quite good results and perform better than the Straw-Man approach in all metrics. However, when trained on complexity or churn features, the models have lower precision and f1-score compared to Straw-Man. These results indicate a significant value of historical data, while also showing that a combination of diverse feature types can be beneficial in capturing the dynamics of workflow reruns.

A similar trend can be observed for job reruns, though with a notable distinction. As shown in Table 8, in this case, models trained solely on historical data consistently achieve the best performance across all metrics, sometimes even outperforming those trained on the full feature set. This indicates that, for job rerun predictions, historical features alone are not only sufficient but also more reliable. In contrast, incorporating complexity or churn features tends to degrade model performance, suggesting that these additional signals may introduce noise rather than contribute meaningful information. These findings highlight a key difference between the two prediction tasks: while predicting outcomes of workflow rerun benefit from a wider set of features, job rerun outcomes are best predicted using historical data alone.

6.3 From Prediction to Developer Actions

The strong performance of our models, particularly the MLP and Decision Tree classifiers, indicates that workflow and job execution outcomes can be predicted with high accuracy. However, accurate prediction alone does not automatically yield practical benefits. To reduce CI/CD waste and improve reliability, model predictions must be translated into actionable policies that guide when and how reruns should proceed.

6.3.1 Rerun Suppression via Threshold Policies. A practical way to operationalize model predictions is through a threshold-based suppression policy: if the predicted success probability is below a chosen threshold τ, the rerun is automatically suppressed (*i.e.,* in a real-world scenario, prompt the developers for manual review). This turns model output into a concrete decision that trades reliability (the missed-success rate, *i.e.,* the proportion of suppressed reruns that would have succeeded) against efficiency (saved waiting time, saved computational time, and avoided unnecessary reruns).

To assess the impact of such policies, we conducted a retrospective simulation on the test sets from Section 6.2. Using our best-performing MLP models, we applied threshold values $\tau \in 0.1, 0.2, \ldots, 0.9$, executing reruns only when the predicted success probability exceeded τ and otherwise suppress their execution. We measured three outcomes: (1) the proportion of unnecessary reruns suppressed, (2) the percentage of waiting and computational time saved (Section 4.3), and (3) the missed-success

rate. Finally, to characterize the practical value of these policies, we analyze the trade-offs between reliability and efficiency by relating (3) to (1) and (3) to (2), separately for workflow and job reruns.

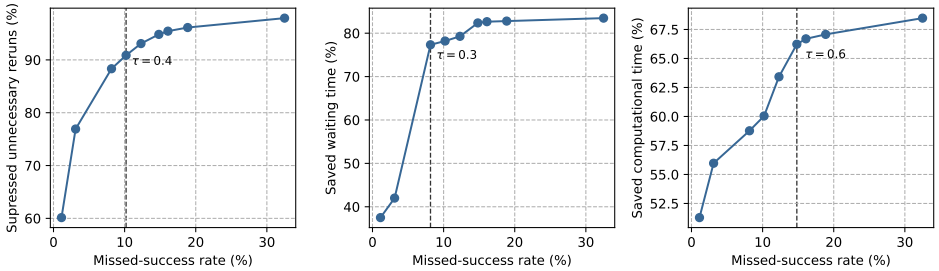

Fig. 11. Trade-offs of threshold-based suppression - Workflow reruns

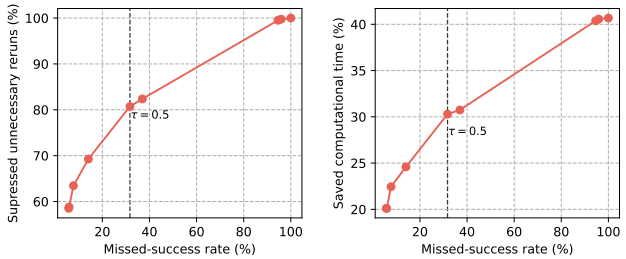

Fig. 12. Trade-offs of threshold-based suppression - Job reruns

6.3.2 Simulation results: Workflow reruns vs Job reruns. Figure 11 and Figure 12 visualize the result trade-offs obtained from the simulation, tracing how efficiency and risk shift as τ increases, with elbow points indicated. Although the trends across suppressed reruns, saved waiting time, and saved computational time follow a similar pattern, the scale and curvature of the trade-off behave quite differently for workflow versus job reruns.

Workflow Reruns: Strong, Early-Gain Trade-offs. For workflow reruns, the trade-off curves are consistently favorable. As τ increases from small values, efficiency metrics rise sharply: unnecessary reruns are suppressed at rates exceeding 90–95%, and saved waiting and computational time commonly reach 70–90% and 60% + before the missed-success rate becomes substantial. Clear elbows emerge around $\tau \approx 0.3$–0.6, where most efficiency gains have already been realized. For example, at $\tau = 0.4$, workflow-level suppression removes 90.87% of unnecessary executions while still preserving 89.8% of successful reruns, resulting in an estimated 2.9 years (78.2%) of waiting time saved and 1.62 years (60.04%) of computational time saved in 2023. Beyond the elbow region, efficiency begins to plateau while the missed-success rate increases more rapidly, indicating diminishing returns. Overall, the nonlinear shape of the curves shows that workflow-level suppression aligns naturally with threshold-based decision-making: moderate τ values deliver disproportionate efficiency gains with limited reliability cost.

Job Reruns: Harsher, More Linear Trade-offs. Job-rerun suppression exhibits a substantially different pattern. Although the proportion of unnecessary reruns suppressed increases steadily with τ, the gains in saved computational time remain comparatively modest, typically peaking

around 20–40% even at higher thresholds. More critically, the missed-success rate grows almost linearly with τ, lacking the early plateau observed for workflows. This makes job-level suppression far less forgiving. Under the same $\tau = 0.4$ used above, job-level suppression eliminates 69.26% of unnecessary executions and saves only roughly 0.44 years (24.59%) of computational time, but does so with a relatively high missed-success rate of 14.1%. These trends suggest weaker separability between redundant and genuinely recoverable reruns at the job level, implying that threshold-based suppression must be tuned more conservatively.

Together, these observations illustrate how prediction can guide actionable CI/CD policies and, importantly, how developers can tune τ based on their priorities—whether optimizing for saved time, reduced compute costs, or minimized disruption to build reliability.

6.4 Discussions

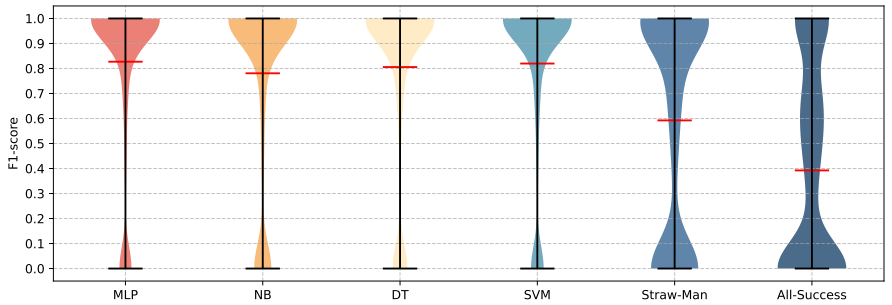

Fig. 13. F1-scores comparison across models @ WF level - Workflow rerun

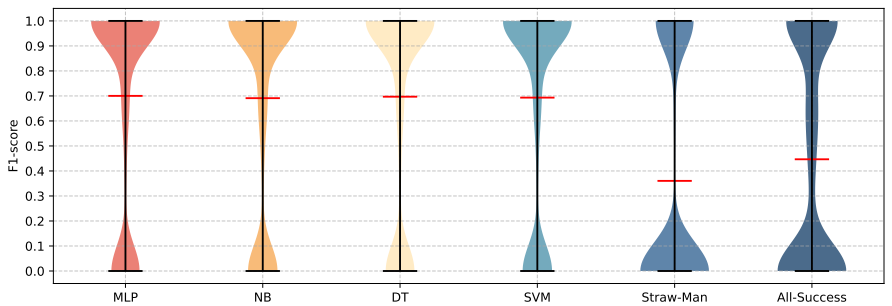

Fig. 14. F1-scores comparison across models @ WF level - Job rerun

6.4.1 Performance at Workflow File Level. Previous comparisons focused on the performance of the models at the level of individual workflow runs. However, since some workflow runs originate from the same WFs, and they share the same complexity and churn data (in cases of reruns, the WFs have not been modified between runs), it is also valuable to examine how the performance varies from different WFs. To achieve this, we grouped all executions from the same WFs and calculated F1-score for each group. Figure 13 and Figure 14 present violin plots of the F1-scores for five models across all workflows, focusing on workflow rerun and job rerun outcome prediction, respectively. In the context of workflow rerun prediction, all four ML-based models consistently

outperform the two baselines (Straw-Man and All-Success), with MLP showing the best overall performance through a higher mean F1-score and reduced variability across workflows. For job rerun prediction, while still outperforming the baselines, the ML models exhibit similar bimodal distributions, indicating that while many workflows are predicted with high accuracy, there remains a substantial subset of challenging cases with which all models struggle. In contrast, the Straw-Man baseline maintains a narrow distribution centered around low F1-scores, underscoring its limited predictive capability, especially for job-level reruns. Overall, these results emphasize the advantage of ML-based approaches over naive baselines and point toward future opportunities to target and enhance predictions on persistently difficult workflow instances.

6.4.2 Impact of History Length on Performance. Given the importance of historical data, we are interested in understanding how different lengths of history would impact the prediction performance. To evaluate for both workflow and job reruns, we selected two sets of workflows from the test set: the top 100 workflows with the highest number of execution records (referred to as "First 100") and the workflows ranked 101–200 by execution count ("Next 100"). For workflow reruns, the "First 100" had an average of 65.5 reruns, while the "Next 100" averaged 17.8 reruns. In terms of job reruns, the "First 100" had an average of 21.1 reruns, compared to 3.2 reruns for the "Next 100".

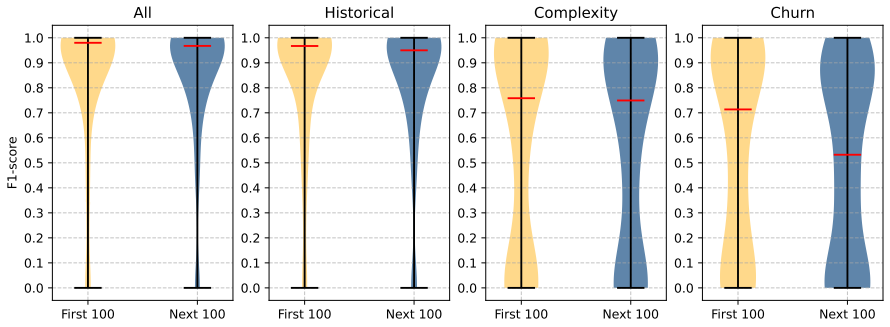

Fig. 15. F1-score comparison across features @ WF level - Workflow rerun

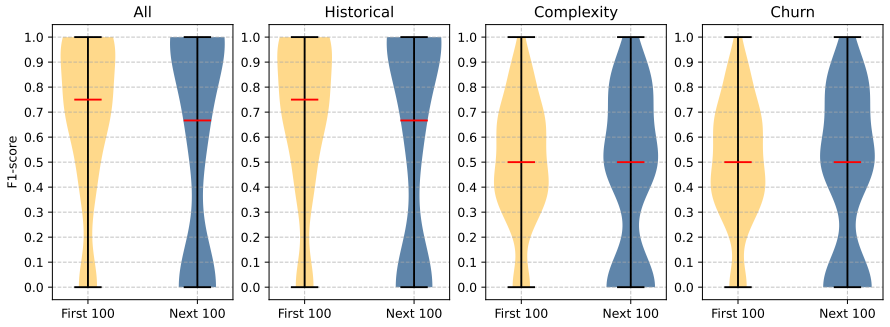

Fig. 16. F1-score comparison across features @ WF level - Job rerun

We analyzed the prediction results for all executions in each workflow achieved by the best performing MLP and represent them with violin plots in Figure 15 and Figure 16. For workflow

reruns, the median F1 scores for both the "First 100" (0.967) and "Next 100" (0.95) in the `Historical` are similar to those in `All` (0.966 and 0.94, respectively); for job reruns, both models trained in `Historical` data and those trained in full data share the same median F1 score (0.75 and 0.67 for "First 100" (0.967) and "Next 100", respectively), further confirming that the historical characteristics contribute strongly to the effectiveness of the models. As for `Complexity` and `Churn` features, the models' F1-scores display a wider spread for both groups of WFs, indicating less consistent performance among different workflows. The corresponding median F1-scores are also significantly lower compared to `All` and `Historical` for both workflow and job reruns, indicating that WF complexity, their branch locations and trigger events have limited contribution.

The median F1-scores are marked in red in Figure 15 and Figure 16. As we can see, across all data categories, models trained on the "First 100" workflows consistently achieve slightly higher median F1-scores than those trained on the "Next 100," for both workflow-level and job-level reruns. This trend suggests that the models perform better when more historical information is available, likely due to the increased amount of training data. However, the observed differences are not substantial, indicating that the MLP classifiers maintain relatively stable performance even when limited historical information is provided.

6.4.3 Why not LLMs? In addition to the methodological considerations discussed in Section 6.1.3, we conducted a small-scale exploratory experiment to evaluate the feasibility of applying large language models (LLMs) to the prediction task. Using GPT-3.5-turbo with few-shot prompting, the model achieved an F1-score of 0.586, with a precision of 0.491 and recall of 0.725. These results are notably lower than those obtained by the traditional machine learning models in our study. While LLMs demonstrate impressive capabilities in natural language reasoning, their advantage diminishes when the input is purely structural and numerical. Moreover, leveraging LLMs introduces considerable computational and energy overheads, which are not justified given the marginal performance gains. Overall, our findings reaffirm that lightweight models provide a more practical and effective solution for this task, maintaining a favorable balance between predictive power, interpretability, and resource efficiency.

Summary RQ_3: To what extent can we predict the outcome of a GHA workflow execution?

We developed and evaluated four ML-based models (Naive Bayes, Decision Tree, SVM, and MLP) to predict the outcomes of GHA workflow reruns. Leveraging a curated dataset comprising over 50k workflow rerun records, we engineered features across three categories: historical, complexity, and churn. Each model was trained using these features both independently and in combination. For workflow rerun outcome prediction, the MLP classifier using the complete feature set achieved the best performance, reaching an F1-score of **90%**. Similarly, for job rerun outcome prediction, the top-performing MLP model attained an F1-score of **70.8%**. In both cases, all ML-based models substantially outperformed the baseline methods, demonstrating the effectiveness of those models across different levels of rerun granularity. Feature analysis revealed that historical data had the highest predictive power, followed by churn and complexity features. These results show the feasibility of using lightweight ML-based models to accurately predict rerun outcomes, potentially reducing unnecessary workflow executions and promoting more efficient CI practices. By translating predicted success probabilities into actionable rerun-suppression policies, these models also support practical decision-making in CI/CD, delivering strong efficiency gains for workflow reruns and highlighting sharper trade-offs at the job level.

7 THREATS TO VALIDITY

The following threats have been identified to validity:

Threats to internal validity concern whether a study accurately reflects a causal relationship between the cause and the effect. In our study, a potential threat is the selection bias in choosing GHA workflows. To mitigate this threat, we selected repositories from diverse domains with varying popularity and activity levels. We also considered workflows from all branches to get a comprehensive view of their usage.

Threats to external validity mainly concern the generalizability of our findings. First, our study focused exclusively on Java projects. While Java is a widely used language, workflows and security practices may vary across languages. Including other languages could improve the generalizability of our findings. Additionally, the workflows analyzed were from the year of 2023, practices of GHA workflow usage may have since evolved. Despite a diverse range of repositories involved in our study, future analysis would be helpful to monitor whether our findings still hold over time.

Threats to construct validity concern the relationship between theory and observation. In our study, we aimed to understand workflow flakiness by manually analyzing its root causes. We employed a saturation strategy, beginning with an initial batch of 30 workflow flakiness sets and incrementally analyzing more until no new causes emerged. Saturation was reached at 50 sets. To ensure more robust results, we ultimately doubled the sample size and analyzed 100 workflow flakiness sets. We acknowledge that some causes may remain uncovered in the whole dataset. However, we believe that the disclosed root causes of workflow flakiness are representative of the majority of the cases.

8 RELATED WORK

8.1 CI Waste

Maipradit *et al.* [54] examined the usage of the "recheck" command, which repeats the build process without updating code, on Gerrit code review platform of the OpenStack community. Their study analyzes the frequency and outcomes of rechecks, and quantifies the computational and time waste associated with unnecessary rechecks. While this study is similar to our RQ1 to a certain extent, unlike their work, we focus on reruns of GHA workflows and analyze a wide range of projects instead of only OpenStack projects. Moreover, our study covers unique GHA feature job rerun.

Weeraddana *et al.* [70] explored wasted CI build time due to updates to unused dependencies in software projects. They found that over half of CI build time related to dependency updates was triggered by unused dependencies, where automated bots, especially "Dependabot", are significant contributors. Similar causes of time wasting on GHA workflows were observed in our study, as presented in Section 5.

8.2 Flaky CI Builds and Tests

The most relevant work in terms of flaky CI behavior was conducted by Durieux *et al.* [9]. They studied the restarted and flaky builds on Travis CI, and discovered that developers restart around 1.72% of the builds, particularly in mature and complex projects, which disrupts the workflow significantly. While their study also categorized the reason behind the build failures (*e.g.,* "No Gemfile found"), we focus on why the workflow execution results are flaky. More studies have been conducted to understand the flaky tests [10, 53, 59]. Luo *et al.* [53] manually analyzed 201 commits from 51 Apache projects that fix flaky tests and presented a taxonomy of the most common root causes of the flaky tests. Eck *et al.* [10] studied the developer's perspective on flaky tests, revealing that they are rather frequent and a non-negligible problem. Parry *et al.* [59] have conducted a survey on the causes, costs, detection strategies, and mitigation of flaky tests. Compared to these

studies, our work focuses on the flakiness of GHA workflow executions, aiming to identify root causes and further provide suggestions on improving the reliability of GHA workflows.

8.3 Build Outcome Prediction

Several approaches have been proposed to predict build outcomes [5, 44, 48]. Chen *et al.* [5] introduced `BuildFast`, which exploits historical data and adapts itself based on previous outcomes to predict build failure. In the evaluation across 20 projects, `BuildFast` enhanced prediction performance by 47.5% in F1-score for failed builds, paving the way for addressing the challenge of long build times and improving cost-effectiveness. Hassan and Wang [44] proposed a build prediction model using TravisTorrent dataset, leveraging build error log clustering and AST level code change data. Their model achieved an average F-Measure exceeding 87% in cross-project prediction scenarios. Jin *et al.* [48] introduced `SmartBuildSkip` to reduce CI costs by predicting and skipping redundant builds. In evaluations, `SmartBuildSkip` saved a median of 30% of builds, demonstrating significant cost savings. Moreover, Abdalkareem *et al.* [2] proposed a rule-based technique to detect CI skip-worthy commits by analyzing over 1,800 real cases, reducing unnecessary builds by 18% and releasing an automated tool (CI-Skipper). They later extended this work using a decision tree model trained on commit features, achieving high accuracy (AUC up to 0.89) and demonstrating cross-project generalizability [1]. Compared to previous work, our research focuses on exploiting ML-based models to predict outcomes of GHA workflow executions. Our best model achieved an F1-score of 90% on the test data, demonstrating the enhanced predictive capabilities of ML techniques.

9 CONCLUSIONS

Our study sheds light on the prevalence of workflow and job reruns in GitHub Actions, revealing the common practice among developers to address failed workflow executions. Despite the wide adoption, our findings highlight their limited success rate in altering workflow outcomes, suggesting a substantial waste of time and computational resources. We also investigated the reasons behind the flaky GHA workflow executions and give practical suggestions on how to avoid unnecessary reruns. Furthermore, we tested ML-based models as a step towards mitigating the resource wastes associated with reruns by predicting workflow run outcomes based on specific features.

In light of the findings and implications in this study, the wastes and flakiness associated with GHA workflow reruns present both pressing concerns and exciting research opportunities for the software engineering community. By devising innovative strategies to optimize workflow execution and minimize resource consumption, researchers and tool builders can contribute to the advancement of CI/CD technologies and the broader field of software engineering. Our future work will focus on tackling these challenges.

REFERENCES

[1] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. 2021. A Machine Learning Approach to Improve the Detection of CI Skip Commits. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2740–2754. https://doi.org/10.1109/TSE.2020.2967380

[2] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. 2021. Which Commits Can Be CI Skipped? *IEEE Transactions on Software Engineering* 47, 3 (2021), 448–463. https://doi.org/10.1109/TSE.2019.2897300

[3] 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. 49–58. https://doi.org/10.1109/ICSE.2015.134

[4] Islem Bouzenia and Michael Pradel. 2024. Resource Usage and Optimization Opportunities in Workflows of GitHub Actions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 25, 12 pages. https://doi.org/10.1145/3597503.3623303

[5] Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. 2021. BuildFast: history-aware build outcome prediction for fast feedback and reduced cost in continuous integration. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 42–53. https://doi.org/10.1145/3324884.3416616

[6] Corinna Cortes and Vladimir Vapnik. 1995. Support-Vector Networks. *Mach. Learn.* 20, 3 (Sept. 1995), 273–297. https://doi.org/10.1023/A:1022627411411

[7] Laura Dabbish, H. Stuart, Jason Tsay, and James Herbsleb. 2012. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. *Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW*, 1277–1286. https://doi.org/10.1145/2145204.2145396

[8] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 560–564.

[9] Thomas Durieux, Claire Le Goues, Michael Hilton, and Rui Abreu. 2020. Empirical Study of Restarted and Flaky Builds on Travis CI. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 254–264. https://doi.org/10.1145/3379597.3387460

[10] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: the developer's perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 830–840. https://doi.org/10.1145/3338906.3338945

[11] 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 *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1330–1342. https://doi.org/10.1145/3510003.3510211

[12] GitHub. 2024. About billing for GitHub Actions. https://docs.github.com/en/billing/managing-billing-for-github-actions/about-billing-for-github-actions#about-spending-limits.

[13] GitHub. 2024. aklivity/zilla - links-check.yml. https://github.com/aklivity/zilla/actions/runs/12758431070/workflow.

[14] GitHub. 2024. apache/dubbo - build-and-test-scheduled-3.2.yml. https://github.com/apache/dubbo/blob/3.3/.github/workflows/build-and-test-scheduled-3.2.yml.

[15] GitHub. 2024. aws/aws-advanced-jdbc-wrapper - remove-old-artifacts.yml. https://github.com/aws/aws-advanced-jdbc-wrapper/blob/main/.github/workflows/remove-old-artifacts.yml.

[16] GitHub. 2024. bytedeco/javacpp - javacpp.yml. https://github.com/bytedeco/javacpp/blob/b3bda856d6130ec1b9fdb255a55e4e42aabc661c/.github/workflows/javacpp.yml.

[17] GitHub. 2024. corretto/corretto-8 - fetch-dev-repo.yml. https://github.com/corretto/corretto-8/blob/develop/.github/workflows/fetch-dev-repo.yml.

[18] GitHub. 2024. deepjavalibrary/djl-serving - llm-integration-p4d.yml. https://github.com/deepjavalibrary/djl-serving/blob/b909f121d06c19f844f03245daed29a1abbe64a8/.github/workflows/llm_integration_p4d.yml.

[19] GitHub. 2024. deeplearning4j/deeplearning4j-examples - example-sanity-check.yml. https://github.com/deeplearning4j/deeplearning4j-examples/actions/workflows/example-sanity-check.yml.

[20] GitHub. 2024. elastic/apm-agent-java - test-reporter.yml. https://github.com/elastic/apm-agent-java/blob/main/.github/workflows/test-reporter.yml.

[21] GitHub. 2024. Events that trigger workflows. https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows.

[22] GitHub. 2024. Example job rerun. https://github.com/sermant-io/Sermant/actions/runs/9279177173.

[23] GitHub. 2024. Example workflow rerun. https://github.com/domaframework/doma/actions?query=+branch%3Amaster+event%3Aschedule.

[24] GitHub. 2024. firebase/firebase-admin-java - nightly.yml. https://github.com/firebase/firebase-admin-java/blob/master/.github/workflows/nightly.yml.

[25] GitHub. 2024. GitHub. https://github.com/

[26] GitHub. 2024. GitHub Actions. https://github.com/features/actions.

[27] GitHub. 2024. mediathekview/MediathekView - nightly.yml. https://github.com/mediathekview/MediathekView.

[28] GitHub. 2024. micronaut-projects/micronaut-spring - graalvm-dev.yml. https://github.com/micronaut-projects/micronaut-spring/blob/c400729726f093921d040cf39e75f1ba1636ba1f/.github/workflows/graalvm-dev.yml.

[29] GitHub. 2024. niumoo/bing-wallpaper - maven.yml. https://github.com/niumoo/bing-wallpaper/blob/main/.github/workflows/maven.yml.

[30] GitHub. 2024. pulumi/pulumi-gcp - upgrade-bridge.yml. https://github.com/pulumi/pulumi-gcp/blob/c642f3251e9b132520f376aaf8a8903723522b7d/.github/workflows/upgrade-bridge.yml.

[31] GitHub. 2024. Rerunning workflows and jobs. https://docs.github.com/en/actions/managing-workflow-runs/re-running-workflows-and-jobsRe-running workflows and jobs.
[32] GitHub. 2024. REST API endpoints for workflows. https://docs.github.com/en/rest/actions/workflows?apiVersion=2022-11-28.
[33] GitHub. 2024. Self-defined actions in repository 'micronaut-projects/github-actions'. https://github.com/micronaut-projects/github-actions.
[34] GitHub. 2024. spring-projects/sts4 - snapshot-e430-eclipse-distro-build.yml. https://github.com/spring-projects/sts4/blob/cd8d0e72a65cd5e49d6ffc3fce48dedf472a28c3/.github/workflows/snapshot-e430-eclipse-distro-build.yml.
[35] GitHub. 2024. Updates on actions' configurations. https://github.com/micronaut-projects/github-actions/commit/67723e0762465909cc0351b19d9b593deb4430a2.
[36] GitHub. 2024. Using GitHub Actions: Actions. https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions#actions.
[37] GitHub. 2024. Using workflows: Creating and managing GitHub Actions workflows. https://docs.github.com/en/actions/using-workflows.
[38] GitHub. 2024. vividus-framework/vividus - framework - gradle.yml. https://github.com/vividus-framework/vividus/blob/master/.github/workflows/gradle.yml
[39] GitHub. 2025. PYPL PopularitY of Programming Language. https://pypl.github.io/PYPL.html.
[40] GitHub. 2025. TIOBE Index for March 2025. https://www.tiobe.com/tiobe-index/.
[41] 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)*. 662–672. https://doi.org/10.1109/SANER53432.2022.00084
[42] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An Exploratory Study of the Pull-Based Software Development Model. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 345–355. https://doi.org/10.1145/2568225.2568260
[43] Greg Guest, Arwen Bunce, and Laura Johnson. 2006. How many interviews are enough? An experiment with data saturation and variability. *Field methods* 18, 1 (2006), 59–82.
[44] Foyzul Hassan and Xiaoyin Wang. 2017. Change-aware build prediction model for stall avoidance in continuous integration. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Markham, Ontario, Canada) *(ESEM '17)*. IEEE Press, 157–162. https://doi.org/10.1109/ESEM.2017.23
[45] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. 197–207. https://doi.org/10.1145/3106237.3106270
[46] Jiangnan Huang and Bin Lin. 2023. CIGAR: Contrastive Learning for GitHub Action Recommendation. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 61–71. https://doi.org/10.1109/SCAM59687.2023.00017
[47] Jiangnan Huang and Bin Lin. 2025. Revisiting Security Practices for Github Actions Workflows. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*. 73–77. https://doi.org/10.1109/ICPC66645.2025.00016
[48] Xianhao Jin and Francisco Servant. 2020. A cost-efficient approach to building in continuous integration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 13–25. https://doi.org/10.1145/3377811.3380437
[49] Omoniyi Babatunde Johnson, Jeremiah Olamijuwon, Zein Samira, Olajide Soji Osundare, and Harrison Oke Ekpobimi. 2024. Developing advanced CI/CD pipeline models for Java and Python applications: A blueprint for accelerated release cycles. *Computer Science & IT Research Journal* 5, 12 (Dec. 2024), 2645–2663. https://doi.org/10.51594/csitrj.v5i12.1758
[50] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21, 5 (01 Oct 2016), 2035–2071. https://doi.org/10.1007/s10664-015-9393-5
[51] Sabrina Klivan, Sandra Höltervennhoff, Rebecca Panskus, Karola Marky, and Sascha Fahl. 2024. Everyone for Themselves? A Qualitative Study about Individual Security Setups of Open Source Software Contributors. In *2024 IEEE Symposium on Security and Privacy (SP)*. 1065–1082. https://doi.org/10.1109/SP54263.2024.00214
[52] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A study on the lifecycle of flaky tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1471–1482. https://doi.org/10.1145/3377811.3381749
[53] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 643–653. https://doi.org/10.1145/2635868.2635920
[54] Rungroj Maipradit, Dong Wang, Patanamon Thongtanunam, Raula Gaikovina Kula, Yasutaka Kamei, and Shane McIntosh. 2023. Repeated Builds During Code Review: An Empirical Study of the OpenStack Community.

arXiv:2308.10078 [cs.SE]
[55] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. On the robustness of code generation techniques: An empirical study on github copilot. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2149–2160.
[56] Kevin P Murphy et al. 2006. Naive bayes classifiers. *University of British Columbia* 18, 60 (2006), 1–8.
[57] Fionn Murtagh. 1991. Multilayer perceptrons for classification and regression. *Neurocomputing* 2, 5 (1991), 183–197. https://doi.org/10.1016/0925-2312(91)90023-5
[58] Doriane Olewicki, Mathieu Nayrolles, and Bram Adams. 2022. Towards language-independent brown build detection. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2177–2188. https://doi.org/10.1145/3510003.3510122
[59] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–74.
[60] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[61] Pooya Rostami Mazrae, Tom Mens, Mehdi Golzadeh, and Alexandre Decan. 2023. On the usage, co-usage and migration of CI/CD tools: A qualitative analysis. *Empirical Software Engineering* 28, 2 (2023), 52.
[62] Dhia Elhaq Rzig, Alaa Houerbi, Rahul Ghanshyam Chavan, and Foyzul Hassan. 2025. Empirical Analysis on CI/CD Pipeline Evolution in Machine Learning Projects. arXiv:2403.12199 [cs.SE] https://arxiv.org/abs/2403.12199
[63] Scikit-learn. 2024. DecisionTreeClassifier. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html.
[64] Scikit-learn. 2024. DummyClassifier. https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html.
[65] Scikit-learn. 2024. GaussianNB. https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html.
[66] Scikit-learn. 2024. MLPClassifier. https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html.
[67] Scikit-learn. 2024. SVC. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.
[68] Todd Sedano, Paul Ralph, and Cécile Péraire. 2017. Software Development Waste. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 130–140. https://doi.org/10.1109/ICSE.2017.20
[69] Yan-Yan Song and LU Ying. 2015. Decision tree methods: applications for classification and prediction. *Shanghai archives of psychiatry* 27, 2 (2015), 130.
[70] Nimmi Rashinika Weeraddana, Mahmoud Alfadel, and Shane McIntosh. 2024. Dependency-Induced Waste in Continuous Integration: An Empirical Study of Unused Dependencies in the npm Ecosystem. *Proc. ACM Softw. Eng.* 1, FSE, Article 116 (July 2024), 24 pages. https://doi.org/10.1145/3660823
[71] Yanming Yang, Xin Xia, David Lo, Tingting Bi, John Grundy, and Xiaohu Yang. 2022. Predictive Models in Software Engineering: Challenges and Opportunities. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 56 (April 2022), 72 pages. https://doi.org/10.1145/3503509