# Why Do GitHub Actions Workflows Fail? An Empirical Study

LIANYU ZHENG, School of Computer Science, Wuhan University, China
SHUANG LI, School of Computer Science, Wuhan University, China
XI HUANG, School of Computer Science, Wuhan University, China
JIANGNAN HUANG, Radboud University, The Netherlands
BIN LIN, Hangzhou Dianzi University, China
JINFU CHEN*, School of Computer Science, Wuhan University, China
JIFENG XUAN*, School of Computer Science, Wuhan University, China

GitHub Actions (GHA), a built-in continuous integration and continuous delivery (CI/CD) service of GitHub, has been widely adopted by developers, streamlining the automation of software development workflows. Despite its popularity, failures frequently occur during GHA workflow executions. Fixing these failures often requires significant human effort, and unsuccessful workflow executions waste computing resources. Understanding the reasons behind workflow failures could provide valuable insights for troubleshooting the existing issues of CI/CD and further improving the development process.

In this paper, we present an empirical study to reveal the reasons behind GHA workflow failures. By manually analyzing 375 failed workflow executions across 260 open-source Java projects, we built a comprehensive taxonomy categorizing the common failure types. The taxonomy was further validated by surveying 151 developers. This study is the first empirical work to analyze GHA workflow failures, bringing valuable knowledge to the field of continuous integration in software engineering. Moreover, our taxonomy and survey results not only underscore the critical need for better tools and practices to mitigate these failures but also indicate the directions to enhance the efficiency and reliability of CI/CD pipelines.

CCS Concepts: • **Software and its engineering → Software development techniques**; • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: GitHub Workflow, Continuous Integration

**ACM Reference Format:**
Lianyu Zheng, Shuang Li, Xi Huang, Jiangnan Huang, Bin Lin, Jinfu Chen, and Jifeng Xuan. 2025. Why Do GitHub Actions Workflows Fail? An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2025), 29 pages. https://doi.org/10.1145/3749371

## 1 INTRODUCTION

As software development becomes increasingly complex and requirements evolve rapidly, manually managing software projects is extremely challenging [1, 18, 56], especially when reviewing,

*Corresponding authors.

Authors' addresses: Lianyu Zheng, lianyuzheng@whu.edu.cn, School of Computer Science, Wuhan University, China; Shuang Li, shuangli.cs@whu.edu.cn, School of Computer Science, Wuhan University, China; Xi Huang, xi_huang@whu.edu.cn, School of Computer Science, Wuhan University, China; Jiangnan Huang, Radboud University, The Netherlands, jiangnan.huang@ru.nl; Bin Lin, Hangzhou Dianzi University, China, b.lin@hdu.edu.cn; Jinfu Chen, School of Computer Science, Wuhan University, China, jinfuchen@whu.edu.cn; Jifeng Xuan, School of Computer Science, Wuhan University, China, jxuan@whu.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1049-331X/2025/1-ART1
https://doi.org/10.1145/3749371

merging, and testing contributions from various developers. These difficulties can lead to significant delays in product deliveries. To address these issues, continuous integration and continuous delivery(CI/CD) has emerged as a fundamental practice in modern software development, aimed at enhancing the quality and speed of software development [18, 19, 59].

The CI/CD process automates various steps across the software development lifecycle, such as building, testing, and deployment, through a pre-defined pipeline [28].

Several CI/CD services, such as Travis CI[1], Jenkins[2], and GitLab CI[3], have gained significant popularity among the software community and have been widely adopted in practice. GitHub Actions(GHA), introduced in 2018, is a built-in CI/CD infrastructure that allows developers to easily create CI/CD pipelines known as GitHub Actions workflow (GHA workflow). Integrated tightly with the GitHub ecosystem, these workflows can be triggered by various events such as pull requests or code pushes. Additionally, GHA supports diverse operating systems and hardware, and offers a generous free tier for open-source projects. Its extensive marketplace of reusable *actions* (custom applications that perform often repeated tasks) also makes it a compelling choice for developers [51]. Within a short span of 18 months, GHA has become the leading CI/CD service on the market [24].

While CI/CD services significantly enhance developer productivity, they can also be time-consuming and resource-intensive. Frequent CI/CD failures, particularly those that take hours or even days to resolve, can negatively impact development efficiency [35]. By understanding the root causes of historical CI/CD failures, development teams can implement preventive actions to streamline the development process.

GHA workflows are executed using two components: actions and scripts. Actions are modular, reusable automation components maintained by GitHub, the open-source community, or third-party providers. They streamline common tasks like repository checkout, environment setup, and application deployment. Defined using the *uses*: directive in workflow YAML files, actions reduce workflow-specific logic and improve maintainability. On average, a workflow makes use of 2.7 distinct actions, and more than 60% of the steps rely on action, underscoring their widespread adoption[16]. In contrast, scripts are custom-written commands executed using the *run*: directive. Unlike actions, scripts are repository-specific and managed directly by developers, introducing greater complexity and variability. GHA differentiates itself from older traditional CI/CD systems like Jenkins and Travis CI through its seamless integration with GitHub repositories and the use of YAML configuration files, which simplifies setup and taps into a vast ecosystem of community-contributed actions. However, dependence on third-party actions from the GitHub Marketplace introduces risks stemming from inconsistent quality, maintenance gaps, and security vulnerabilities. Issues such as Poor documentation, infrequent updates, or sudden deprecations can lead to unexpected failures and compromise pipeline stability. These challenges highlight the need for rigorous vetting, continuous monitoring, and empirical studies to better understand and mitigate the unique failure modes of GHA.

Existing research on CI/CD failures has primarily focused on compilation [66], build [39, 62], and test failures in traditional CI services like Jenkins [6, 12]. For instance, Lou et al. [39] conducted a comprehensive study analyzing 1,080 Stack Overflow posts related to Maven, Gradle, and Ant, developing a fine-grained taxonomy of 50 build failure symptoms and identifying common fix patterns. Their findings revealed that 67.96% of build issues are resolved through plugin and dependency modifications. Similarly, Rausch et al. [49]examined CI build failures in 14 Java-based

[1] https://www.travis-ci.com
[2] https://www.jenkins.io
[3] https://about.gitlab.com/solutions/continuous-integration

open-source projects using Travis CI. However, these studies rely on older CI services, use coarse-grained error categories such as compilation, dependency, and Git issues, and derive insights from limited project samples, making them insufficient to address the unique challenges faced by GHA users. Given GHA's widespread adoption, there remains a significant lack of research on its platform-specific failures. To our knowledge, no comprehensive empirical study has yet explored the root causes of CI / CD failures specific to GHA. To address this gap and overcome these limitations, we present a detailed empirical study of GHA workflow failures, combining analysis of failed workflows on GitHub with insights from a developer survey.

We sampled 375 GHA workflow execution failures and manually analyzed the logs to identify the root causes of these failures. Using the card-sorting method [57], we developed a taxonomy of failure categories, resulting in a total of 16 different types of failures. Subsequently, we gathered developer feedback on our identified failure categories through an online survey. We distributed the survey to developers actively using GHA in open-source projects, collecting 151 valid responses. These responses validate our taxonomy and provide new insights into the specific challenges developers face when troubleshooting workflow failures and the support they need. These insights shed light on potential solutions to address workflow failures and indicate new research directions.

To support replication and further research, we have made the original datasets publicly available[4]. The datasets include the replication-labeling file, which contains the final categorization results from our manual investigation, detailing each failed job's unique identifier, project, and workflow names, failure step, manually identified root cause, and categorized failure type. Additionally, the replication-surveyAnswers file presents responses from GitHub Actions practitioners, including survey questions and answers that offer insights into developers' experiences with workflow failures. The package also includes a scripts directory with code used to extract logs from failed workflows and a data directory containing the necessary data files for log extraction and analysis. The main contributions of our work are as follows:

- We conducted the first extensive empirical analysis on GHA workflow failures, identifying and categorizing the types of failures developers encountered during CI using GHA.
- We provided both qualitative and quantitative evidence of the reasons behind GHA workflow failures, integrating examples from real-world projects and feedback from developers.
- We summarized the challenges developers encountered when dealing with GHA workflow failures and the support they would need to aid this process.
- We provided the original datasets and scripts available in a replication package, enabling further studies on CI/CD failure diagnosis.

The remainder of the paper is structured as follows: Section 2 discusses the related work. The research questions and the study design are presented in Section 3. Section 4 reports the results of our study. Section 5 discusses the analysis of the differences between GHA failures and previous CI/CD tools. The threats to validity are presented in Section 6. Section 7 concludes this work with future research directions.

## 2 RELATED WORK

In this section, we present the related work on the use of GitHub Actions and studies on CI/CD failures.

### 2.1 GitHub Actions

GitHub Actions (GHA) has rapidly emerged as one of the most preferred platforms for implementing Continuous Integration and Continuous Deployment (CI/CD) pipelines [24]. GHA offers

[4]https://github.com/zhengly1/workflow_failure

a range of functionalities that facilitate automated software testing, building, and deployment. According to Kinsman et al. [36], GHA has received a positive reception among software developers, indicating its significant impact on modern development practices.

Currently, researchers have conducted numerous studies to understand the use of GHA in practice. Saroar et al. [55] investigated the purposes, usage processes, and challenges faced by developers using GHA. Their study sheds light on the practical experiences of developers, highlighting common issues and areas for improvement. Cardoen et al. [9] further contributed to this body of research by collecting and publicly releasing a comprehensive dataset of over 160,000 commit histories of workflow files from more than 32,000 public GitHub repositories. This dataset covers over 1.5 million workflow file versions, providing valuable insights into the usage patterns and evolution of CI/CD workflows.

Efforts have also been taken to facilitate the adoption of GHA. Zhu et al. [68] introduced ACTIONSREMAKER, a tool designed to reproduce GHA workflow runs accurately. ACTIONSREMAKER takes as input the desired workflow job to reproduce and outputs a Docker image with the exact environment, software repository, and build script. This tool is particularly useful for debugging and ensuring consistency across different development environments. Moreover, Mastropaolo et al. [40] proposed GH-WCOM (GitHub Workflow Completion), an innovative approach that leverages Transformer models to automatically complete GHA workflows. This approach enhances developer productivity by reducing the manual effort required to compose workflow configurations.

Other researchers have looked into the quality improvement of GHA workflows. For example, Bouzenia et al. [8] conducted an empirical study on resource utilization and optimization opportunities within the GHA platform. Their research aims to enhance the resource efficiency of CI/CD pipelines by identifying areas where resources can be better managed.

Security issues within GHA have also been a popular research focus. Koishybayev et al. [37] defined four security properties essential for securing CI/CD platforms against supply-chain attacks. They also proposed strategies that can be implemented as part of GHA workflows to mitigate security vulnerabilities. These contributions are crucial for protecting software projects from potential threats and ensuring the integrity of the development process. Based on a dataset of nearly one million GHA workflows from over 22,000 repositories between November 2019 and September 2022, Decan et al. [16] conducted an empirical study focused on two main research goals. First, they examined the prevalence and evolution of reusable *actions*, revealing common usage patterns and frequent updates, as well as discrepancies in versioning practices. These findings suggest the need for more standardized versioning guidelines to ensure the reliability of the *actions*. Second, they investigated the outdatedness of GHA workflows, showing that most workflows used outdated *actions* despite opportunities to update. This outdatedness increases vulnerability risks and underscores the necessity for better management and policies to keep workflows up-to-date.

Despite the widespread adoption of GHA, few studies have looked into the reasons behind the failures of GHA workflows. Our study fills a critical gap by providing an empirical analysis specific to GHA, offering a detailed taxonomy of GHA workflow failures, and integrating developer perspectives to ensure the findings are both comprehensive and applicable to real-world scenarios.

### 2.2 CI/CD Failures

CI/CD systems have become a cornerstone of modern software development, facilitating automated testing and the integration of code changes. Despite their benefits, CI/CD systems are not immune to failures, which can significantly disrupt the development workflow. A substantial body of research has been dedicated to understanding the root causes, effects, and mitigation strategies of CI/CD failures. Rausch et al. [49] provided a comprehensive empirical analysis of CI usage in open-source projects, examining the frequency and impact of CI failures and offering valuable

insights on how factors such as task complexity and build tools influence build outcomes. The results from this study substantially advance our understanding of how development factors affect build results. Pinto et al. [46] conducted a user survey with 158 CI users, revealing that overconfidence in CI systems, a lack of testing culture, and the promotion of a fast development cycle are also causes of failure. Zhang et al. [66] made a significant contribution by exploring debugging practices for build failures in CI. Their large-scale empirical study analyzed 6,854,271 CI builds from 3,799 open-source Java projects, summarizing fix patterns for the ten most common compiler error types. Ziftci et al. [69] introduced a novel technique called CULPRIT FINDER to identify changes that induce failures in large tests. This technique uses heuristics to automatically filter and rank change lists that might have introduced the regression, helping developers identify the root cause. Vassallo et al. [62] examined the adoption of CD practices at ING Nederland, focusing on how agile methods and DevOps teams were integrated within a large financial organization. Their research emphasized the significance of automated testing, refactoring, and technical debt management while also discussing the challenges and benefits of CI and its impact on developer productivity. Similarly, Beller et al. [6] analyzed test-induced build failures in Travis CI, identifying failure patterns and their implications for CI reliability, providing a complementary perspective on test-related challenges. Lou et al. [39] investigated build failure resolution by analyzing 1,080 Stack Overflow posts related to Maven, Ant, and Gradle. They developed a detailed taxonomy of 50 symptom categories and identified common fix patterns, finding that 67.96% of issues were resolved by modifying build script code for plugins and dependencies. Their work extended previous studies on build failures by offering a fine-grained classification of symptoms and practical fix insights, complementing automated fixing techniques and highlighting the challenges developers faced with nonintuitive solutions.

Other recent works have leveraged various techniques to predict the outcomes of CI/CD builds. Xia and Li [63] explored the performance of six models to predict CI build outcomes for 126 open-source projects. Saidani et al. [53] introduced a novel Multi-Objective Genetic Programming (MOGP) approach to predict CI build outcomes. This approach adapts the Non-dominated Sorting Genetic Algorithm (NSGA-II) with a tree-based solution representation to generate rules from historical CI build data using two competing objectives: the probability of detection and the probability of false alarms. More recently, Koishybayev et al. [54] introduced deep learning-based approaches to predict CI failures. Their research demonstrates the effectiveness of predictive models in identifying potential failures before they occur, enabling proactive measures to prevent disruptions in the CI pipeline.

Overall, these studies provide valuable insights into CI failures, offering a range of solutions from debugging techniques to predictive models. They underscore the complexity of maintaining robust CI systems and highlight the ongoing need for research to address the evolving challenges in this critical area of software engineering. While previous studies have explored CI failures in general, our work specifically targets GHA, the dominant CI platform nowadays.

## 3 STUDY DESIGN

This study aims to disclose the reasons behind the failures of GHA workflow executions and provide valuable insights on how to improve the CI/CD process on GitHub. The context of this study involves two primary sources of data: (1) 375 failed GHA workflow executions along with corresponding logs from 260 Java projects, and (2) 151 responses from developers who are experienced in GHA gathered through online surveys. The study aims to answer the following research question:

**What are the root causes of GHA workflow failures?**

Our study focuses on developers and researchers working with GHA, aiming to uncover the root causes of workflow failures and the challenges encountered in practice. To investigate these questions, we adopted a two-phase approach. First, we manually analyzed 375 failed GHA workflow executions and their associated logs from 260 open-source Java projects. Based on this analysis, we developed a comprehensive taxonomy categorizing the root causes of workflow failures. Second, we conducted an online survey targeting 7,657 developers with experience in GHA workflows, receiving 151 valid responses. The survey validated our taxonomy and provided insights into the difficulties developers face in troubleshooting GHA failures, as well as their suggestions for improving the CI/CD process on GitHub.

Figure 1 depicts the overview of our two-phase approach, including manual analysis of failed GHA workflows and a survey with developers.

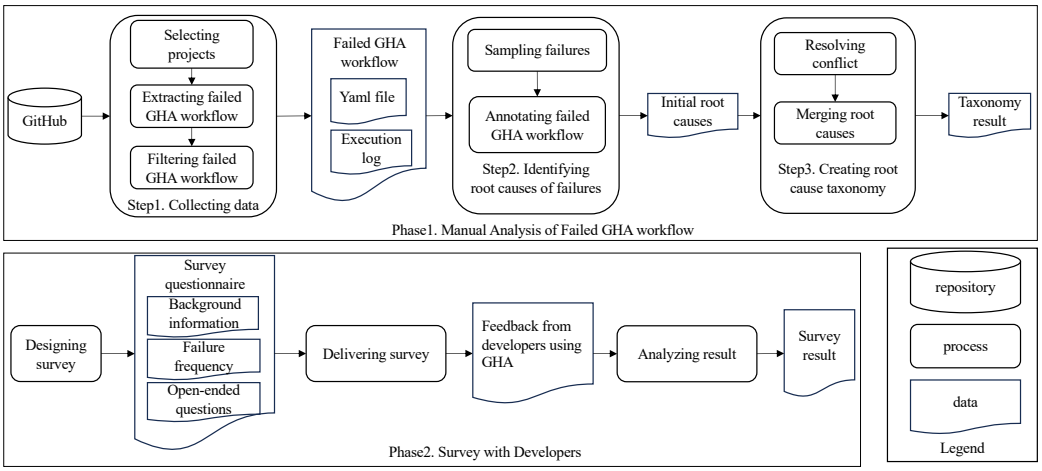



Fig. 1. An overview of our approach

### 3.1 Manual Analysis of Failed GHA Workflows

To understand what causes GHA workflow failures, we performed an exhaustive manual analysis involving 375 failed workflow executions.

*3.1.1 Collecting Data.* As workflow files and their execution logs are essential to diagnose the root cause of execution failures, our study mainly focused on open-source projects. To select the projects, we leveraged the SEART GitHub Search Engine developed by Dabic et al. [14], which is widely used in empirical software engineering research for sampling software repositories. More specifically, we applied the following selection criteria:

- The project is developed in Java, as Java is one of the most used and studied programming languages [61].
- The project has at least 100 stars, such that toy projects can be excluded.
- The project should have more than 100 commits to ensure there is enough history data to analyze.
- The latest commit should be made within three months before the data collection date (November 23, 2023) to comply with GitHub's 90-day log access limitation [23].

These criteria led to a list of 4,312 Java projects. We further excluded those projects that did not use GHA, narrowing our focus to 1,732 projects. Our dataset includes 329 forked projects. Since

forked projects do not inherit GHA workflow execution records from their upstream repositories, their inclusion does not introduce duplicate workflow data. Additionally, we applied a filtering criterion based on star counts to exclude personal forks, further reducing redundancy. Through rigorous verification, we have confirmed the absence of duplicate projects in our dataset. We then employed the GitHub REST API[5] to retrieve the failed executions in the project workflows. As logs are the main source for identifying the reasons behind workflow failures, we excluded GHA workflow executions that do not contain any logs, ultimately obtaining 15,355 valid items. This comprehensive dataset forms the basis for our analysis.

*3.1.2 Identifying Root Causes of GHA Workflow Failures.* Given the substantial volume of logs, it is impractical to review all of them manually. To address this, we adopted a sample size formula for an unknown population [15], ensuring a 95% confidence level with a ±5% margin of error, a standard combination of statistical rigor and practical feasibility in empirical studies [27, 29, 42, 60, 65]. A 95% confidence level means that if we repeatedly sample from the population using the same sampling and statistical procedure, 95% of the resulting confidence intervals contain the true population parameter. The ±5% margin reflects a balance between analytical precision and operational feasibility, ensuring an acceptable level of estimation accuracy. The results indicated that 375 logs would be needed for the analysis. We randomly selected 375 logs from the 15,355 logs collected in the previous step. From the 375 randomly selected logs, we identified a diverse set of actions(from both the marketplace and third-party developers), reflecting the varied tasks within GHA workflows. These actions span multiple functionalities, including environment setup (e.g., actions/setup-java for Java configuration), build processes (e.g., github/codeql-action/autobuild), testing and quality assurance (e.g., github/codeql-action/analyze for security analysis), and release automation (e.g., softprops/action-gh-release for GitHub Releases). Workflow utilities like actions/checkout and notification tools such as rtCamp/action-slack-notify further illustrate their breadth. In total, 131 unique actions were observed, with some appearing frequently (e.g., actions/checkout in nearly all workflows) and others tied to niche tasks (e.g., reactivecircus/android-emulator-runner for Android testing). This distribution underscores the heterogeneity of CI/CD needs across the 260 Java projects, ranging from general-purpose setup and checkout actions to specialized tools for security, deployment, and platform-specific testing, highlighting the adaptability and complexity of GHA in practice.

To categorize the reasons behind GHA workflow failures, we employed a card sorting methodology [57], a well-established technique for organizing qualitative data into meaningful clusters through iterative grouping and refinement. This approach, devoid of predefined categories, allowed theories to emerge collaboratively from the data itself. Specifically, each of the 375 collected logs was independently reviewed and annotated by two authors to enhance reliability and mitigate individual bias. Given the geographical dispersion of our team, we developed a custom web application to streamline and standardize the annotation process (depicted in Figure 2).

The web application facilitates systematic log analysis by randomly assigning each log to two annotators and presenting critical details for annotation, including the job name, workflow name, link to the workflow, the step where an error occurred, and a downloadable log file. Annotators were tasked with assigning root cause labels to each log, with the flexibility to apply multiple labels when multiple failure reasons were evident. To ensure consistency across annotations, we encouraged the reuse of previously established labels whenever applicable. However, to maintain specificity and accuracy, annotators were required to articulate the precise root cause—e.g., for compatibility-related failures, they had to specify the incompatible software component and its environment. When existing labels inadequately captured an issue, annotators created new ones,

[5] https://docs.github.com/en/rest

which were then added to the reusable label pool. The application tracked each annotator's labels and permitted revisions, while blinding annotators to their counterparts' assignments to prevent bias and encourage independent judgment.

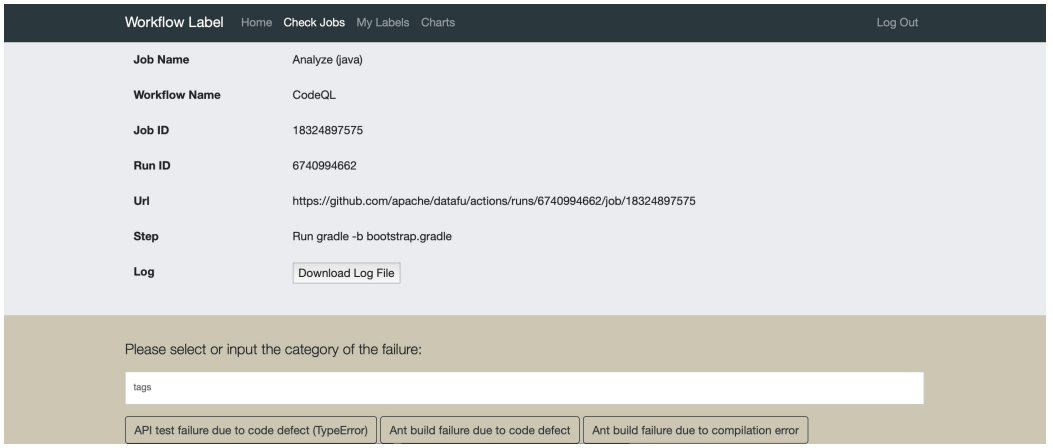



Fig. 2. Website used for manually inspecting GHA workflow failure logs.

To ensure consistent and accurate identification of failure causes, we followed a systematic procedure inspired by established qualitative analysis practices [17, 49, 57]. First, we selected a trial batch of 30 logs for calibration. Six authors participated in this phase, each bringing over five years of software development experience and three to seven years of hands-on expertise with CI/CD tools, including Jenkins, Travis CI, and GitHub Actions. Notably, all annotators possessed at least one year of direct experience working with GHA workflows prior to this study. Additionally, they had prior experience in log analysis, further equipping them for the task. Initially, each annotator independently labeled the 30 logs. Then, during a video conference, all annotators collaboratively reviewed the 30 logs, setting aside initial disagreements to scrutinize labels, resolve discrepancies, and refine the annotation guidelines. To pinpoint error information within the logs, we adopted a keyword-based approach, leveraging a set of indicative terms—"error", "exception", "invalid", "failure", and "fault" [43]—to locate critical failure signals. These keywords enabled the annotators to quickly identify error messages (for example, spotting "exit code 1"), after which they examined the surrounding context to deduce precise root causes. This collaborative exercise, which combined keyword-driven localization with contextual analysis, was instrumental in aligning interpretations, establishing clear labeling criteria, and standardizing the annotation process across all annotators.

Following the initial trial, the six authors proceeded to annotate the remaining 345 logs. To reduce fatigue and ensure consistency, conflict resolution meetings were scheduled after every 100 logs. During these meetings, all annotators collaboratively discussed any conflicting annotations, drawing on their collective expertise in debugging CI/CD pipelines and interpreting failure logs. We reviewed relevant log information and reached a consensus through careful deliberation. This iterative approach to conflict resolution resulted in high interrater reliability, as demonstrated by a Cohen's Kappa coefficient of 0.708 across the dataset, which signifies substantial agreement according to established benchmarks [41]. After resolving all the conflicts and merging highly similar labels, we ended up with 170 unique ones.

*3.1.3 Creating Taxonomy of Root Causes for GHA Workflow Failures.* The six authors, leveraging their extensive expertise in software development and CI/CD practices, collaboratively transformed the 170 unique labels into a structured taxonomy through an iterative card sorting process [57]. The primary objective was to consolidate labels with overlapping meanings and organize them into clear, coherent categories.

The first author created the first draft of the taxonomy. This initial taxonomy was guided explicitly by three carefully defined criteria. First, semantic similarity was considered, grouping labels with overlapping or closely related meanings. For instance, labels such as "missing dependency" and "dependency not found" were grouped under a broader category related to dependency resolution issues. Second, the specific scenarios of CI/CD were taken into account, such as compile, build, and test, ensuring the taxonomy aligned closely with practical workflow scenarios. This classification was guided by insights from prior studies on CI/CD failures [22, 49]. Third, labels were categorized based on their underlying causes, distinguishing between project-related failures (e.g., compilation errors, test failures) and workflow-related failures (e.g., Unable to resolve action due to version Issue, environment setup failures).

Following this initial draft, the remaining five authors independently reviewed the proposed taxonomy through shared collaborative documents. They provided detailed feedback, suggested refinements (such as merging or splitting clusters), and raised specific questions or concerns about certain groupings through detailed comments. These comments and suggestions were thoroughly discussed during multiple rounds of collaborative meetings. In these discussions, authors systematically addressed each discrepancy and collaboratively adjusted the groupings until achieving a unanimous consensus. Through this iterative refinement, overlapping labels were consolidated, and clusters were progressively refined for enhanced coherence, precision, and usability. This extensive collaborative effort culminated in a refined taxonomy consisting of 16 clearly defined root cause categories for GHA workflow failures, offering a comprehensive, precise analytical framework to effectively support failure analysis.

### 3.2 Survey with Developers

To validate our taxonomy and gather feedback from developers who are experienced with GHA, we conducted a survey with three primary goals:

(1) to understand how frequently developers encounter each type of GHA workflow failure observed during our investigation.
(2) to identify any failure types we might have missed during our manual inspection.
(3) to enrich the results by asking for additional insights on the major difficulties developers face when dealing with GHA workflow failures and how these failures could be prevented.

*3.2.1 Designing Survey.* The survey is structured into three sections:

- **Background Information** consists of general questions about the background and experience of the respondents.
- **Failure Frequency** assesses the prevalence of GHA workflow execution failures. In particular, we asked developers to indicate how often they encounter each type of failure defined in our taxonomy using a 5-level Likert scale (i.e., "Most of the time", "Often", "Sometimes", "Rarely", and "Never"). An "I don't know" option is also provided.
- **Open-Ended Questions** gathers direct feedback on challenges and prevention strategies. To ensure the integrity of our taxonomy, we ask respondents to report any other types of failures they had encountered that were not mentioned in the survey. Finally, we invite developers to comment on how the GHA workflow failures could be prevented or solved.

*3.2.2 Delivering Survey.* Given the very specific topic studied in this work, it is crucial to select appropriate survey respondents. We selected all developers who modified files in the 'github/workflows' directory across 1,732 projects in the dataset. Our selection was based solely on whether developers contributed to workflow files, without considering the frequency, recency, or extent of their contributions. This approach ensures that survey participants have at least a basic understanding of GHA workflows, since they have directly interacted with workflow files in open source projects on GitHub.

We distributed the survey to 7,657 selected developers, providing a 14-day window for responses. From these developers, we received 151 valid replies, resulting in a response rate of 1.97%. Several factors likely influenced this relatively low response rate. Firstly, our dataset included multiple email addresses for some developers, indicating that the actual response rate per unique individual might be higher than reported. Secondly, outdated or inactive email addresses could have contributed to lower engagement. Additionally, our selection criteria required only minimal familiarity with GHA, resulting in a subset of developers with limited workflow development experience who may have been less inclined to participate. Responding to surveys also demands time and effort, and developers might have prioritized other commitments, particularly if they perceived no immediate benefit. Despite these challenges, the collected responses provided valuable qualitative insights that complement our empirical analysis.

The majority of respondents identified as developers (68.2%), followed by technology leaders (20.5%) and architects (5.3%). The remaining respondents included test engineers, data scientists, project leads, and individuals who chose not to disclose their roles.

Table 1. The respondents' years of development

| Professional software development experience | <3 years | 3-5 years | 6-10 years | >10 years |
|---|---|---|---|---|
| Number of respondents | 24 | 32 | 32 | 63 |

Table 2. The experience with GHA of the survey respondents

| GHA workflows experience | Novice | Advanced Beginner | Competent | Proficient | Expert |
|---|---|---|---|---|---|
| Number of respondents | 9 | 43 | 44 | 43 | 12 |

Tables 1 and 2 illustrate the respondents' experience in software development and their familiarity with GHA. While the survey covers the entire spectrum of experience, the distribution reveals a different trend: 62.9% of respondents have over 5 years of software development experience, yet only 36.4% are proficient with GHA. This disparity may be attributed to the relatively recent introduction of GHA, with experienced developers still in the process of mastering this tool.

## 4 RESULTS

During our manual analysis, we identified 16 different types of root causes of GHA workflow failures, which can be further categorized into project-related and workflow-related. Project-related failures arise from issues within the project's source code, dependencies, or configuration that obstruct successful workflow execution. These typically manifest as compilation errors, missing dependencies, or failing test cases. A key criterion for identifying project-related failures is whether the issue remains when the same code is executed in a local environment, independent of any GHA components. In contrast, workflow-related failures are due to misconfigurations, limitations, or

external dependencies within the GHA environment itself. Such failures occur independently of the project code and often result from incorrect workflow definitions, missing secrets or permissions, unexpected changes in third-party services, or resource constraints in GHA runners. The primary criterion for classifying a failure as workflow-related is that it only occurs within the GHA environment and would not be present if the code were run outside of it. We discuss these two categories of failures in Section 4.1 and Section 4.2, respectively. To clearly present our findings and highlight the insights, we use the following notations:

- ℹ provides a real-world example of GHA workflows that fail due to reasons belonging to the current category.
- 👤 illustrates survey respondents' experience or opinions related to the failure. Each respondent is represented by R#.

### 4.1 Failures Due to Project-related Factors

During our annotation process, we identified 315 workflow failures caused by project-related factors, accounting for 84% (315/375) of the total cases. The reasons behind these failures can be categorized into eight distinct subcategories, which are represented in Figure 3. These eight types of failures are related to compilation, quality, and testing. The numbers in the blue squares indicate the frequency of each type of failure in our manual analysis. Below, we present a detailed description of these categories, supplemented by real-world examples and insights gathered from surveys of developers actively involved in GHA workflow activities.

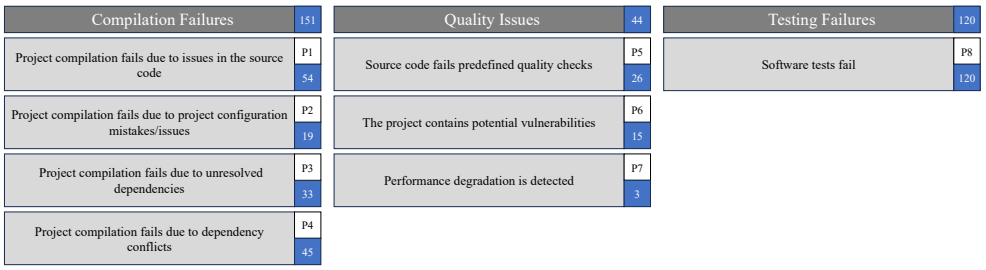



Fig. 3. Taxonomy of failures resulting from project-related factors.

*4.1.1 Compilation Failures.* Compilation failures occur when developers introduce defects in the source code or project configurations. This type of failure can be further broken down into four subcategories.

**P1: Project compilation fails due to issues in the source code**. This is the second most common failure cause in our study, accounting for 14.4%(54/375) of the total cases. When a developer automates the construction, testing, and deployment of projects through GHA workflow, the first step is often compiling the source code. If there is a problem within the source code, the workflow task will fail. Compilation failures can stem from various issues. Syntax errors, such as missing semicolons, unclosed brackets, or incorrect use of language constructs, are common culprits[50]. Additionally, logical errors, where the code does not conform to the expected logic or rules of the programming language, can also lead to compilation failures. Other typical issues include undeclared variables, type mismatches, and incorrect function calls. Incomplete or incorrectly implemented interfaces and missing imports or dependencies further contribute to these failures. Each of these errors interrupts the compiler's ability to convert the source code into executable binaries, thereby halting the entire build process. These compilation failures indicate potential flaws in the code that need to be addressed to ensure a functional and reliable software product.

ℹ For example, in the workflow execution "Android CI #236" [3], the failure occurred when developers tried to directly access the `instructions` field in the `Route` class and the `text` and `sign` fields in the `Instruction` class, which are all private.

**P2: Project compilation fails due to project configuration mistakes/issues**. Project build failures often occur due to mistakes or issues in project configuration. These issues can range from specifying incorrect paths to setting wrong environment variables. When paths to dependencies or modules are incorrectly defined, the build process cannot locate necessary files, resulting in errors. Similarly, incorrect environment variables can lead to misconfigured settings that disrupt the build process. Such configuration errors are common and can significantly hinder development progress if not identified and resolved promptly. Properly managing project configurations is crucial to ensure a smooth and successful build process.

ℹ For instance, in workflow execution "CodeQL #5112" [31], the build process failed because Maven was unable to locate the specified JDK configuration. This issue could be due to an incorrect path, a missing JDK installation, or restricted access to this location.

**P3: Project compilation fails due to unresolved dependencies**. Unresolved dependencies often lead to project build failures because the compiler cannot retrieve the necessary libraries or modules required for the code to function.

These issues may arise if the specified version of a library is not available in the repository or if network issues prevent the dependency from being downloaded. Additionally, misplaced, incorrectly named, or permission-restricted dependencies can also lead to build failures.

ℹ For example, in the workflow execution "Build#2599" [11], the error message indicates that the Gradle build could not find the *cc.tweaked:cobalt:0.8.0-SNAPSHOT* dependency.

🗣 R149: "*The majority of workflow failures I have had are due to upstream Grable dependencies (plugins) dropping out of support, so they become no longer available on their maven. There has also been a recent uptick in issues with jetpack due to (what I believe to be) the Crowdstrike situation.*"

**P4: Project compilation fails due to dependency conflicts**. Dependency conflicts arise when different parts of a project or different projects within a larger system require different versions of the same library. In our study, this type of failure occurs more frequently than unresolved dependencies. These conflicts can cause significant issues during the build process, as the system may be unable to determine which version of the dependency to use. This is a common issue in complex projects with multiple dependencies, where transitive dependencies introduce varying versions of the same library. Such conflicts can lead to class version mismatches, runtime errors, or build failures.

ℹ For example, in the workflow execution "Mobile-Wallet CI[Master/Dev] #661" [44], during the build process, an error occurred indicating an incompatibility between the version of Java being used and the version required by the Android Gradle plugin. Specifically, the error message stated: "Android Gradle plugin requires Java 17 to run. You are currently using Java 11.". This discrepancy necessitates an update to the Java version in the build environment to ensure compatibility and successful execution of the build process.

*4.1.2 Quality Issues.* Quality issues in software development refer to problems that occur when the code does not meet established standards and guidelines[7, 34]. 12% (44/375) of the analyzed cases fall into this category, which can be further divided into three subcategories.

**P5: Source code fails predefined quality checks**. Quality checks are essential in software development, ensuring that the code meets certain standards and maintains a level of consistency and reliability. These checks typically cover several critical areas: code style conventions, documentation standards, and test coverage requirements. Code style conventions ensure that the codebase remains uniform and readable. Inconsistent indentation, improper naming conventions, or varying coding styles can lead to confusion and errors, making the code difficult to read and maintain. Documentation standards are another crucial aspect of quality checks. Proper documentation, including accurate and complete comments, is essential to understand the purpose and functionality of the code. Test coverage requirements ensure that a significant portion of the code is tested, reducing the likelihood of bugs and errors. Adequate test coverage is vital for verifying that the code performs as expected and meets the specified requirements. When code fails to meet these predefined quality checks, it can lead to build failures.

> ℹ For example, in the workflow execution "Android CI#194" [13], the error message indicates that the execution of the task "`:app:lintDebug`" failed.
> Lint is one of the most widely used static code analysis tools to identify and correct stylistic issues. The issues identified by linting tools do not necessarily impact the functioning of software projects. However, when developers have high standards for code quality and would like to strictly enforce the quality rules, passing the check of linters is mandatory in CI workflows.

> ℹ Quality check not only involves stylistic checks, but also enforces test coverage requirements sometimes. For example, in the workflow execution "build #3487" [64], the log indicates that the build for the *ogbook-spring* module failed as the coverage reported by the JaCoCo Maven plugin did not meet the pre-defined value.
> Specifically, the error message shows that the line coverage of `ReflectionUtil` is 0.5, while the expected minimum is 1.0".

**P6: The project contains potential vulnerabilities**. Identifying and addressing potential vulnerabilities is essential for ensuring the security and robustness of a project. Potential vulnerabilities can arise from insecure coding practices, outdated dependencies, and misconfigured settings. These vulnerabilities can be exploited by malicious actors, leading to data breaches, unauthorized access, and other security incidents.

> ℹ For example, in the workflow execution "Security vulnerability scan #472" [30], the output log indicates that the build failed due to vulnerabilities identified by the *OWASP Dependency-Check* plugin. In this case, each of the two problematic jar files contains one vulnerability.

**P7: Performance degradation is detected**. Performance degradation is a significant concern in software development, as it can affect the user experience and the overall efficiency of the application. Performance issues can arise from inefficient algorithms, resource-intensive operations, and suboptimal configurations, leading to slower response times, increased resource consumption, and reduced scalability. Addressing these issues involves conducting thorough performance testing and profiling, identifying performance bottlenecks, and optimizing code and configurations to enhance performance. Regular performance monitoring and tuning are essential to ensure that the application continues to perform well as it evolves and scales.

> ℹ For example, in the workflow execution "Ion Java performance regression detector #251" [2], the log indicates that the build process failed due to benchmarking results that did not meet the expected performance criteria.

While the log did not specify the exact performance differences, the process exited with code 1, indicating that the benchmark results were outside the acceptable range.

*4.1.3 Testing Failures.* This category encompasses all failures related to failed tests. This is the most common reason for GHA workflow failures, accounting for 32% (120/375) of the cases.

**P8: Software tests fail**. Software tests are an integral part of the development process, ensuring that the code meets the required functionality requirements. In our study, we identified issues caused by both unit testing and integration testing.

Unit testing typically targets individual functions in isolation, and integration testing assesses interactions among multiple modules or components, often making its failures more challenging to diagnose. Among the 120 test failures, we categorize them into two subtypes: unit testing failures, accounting for 102 instances (85% of P8), and integration testing failures, comprising 18 instances (15% of P8). Unit testing failures arise predominantly from defects within isolated code units, such as flawed logic, overlooked boundary conditions, or assertion mismatches[26, 52, 67], and are generally easier to pinpoint and resolve due to their limited scope. In contrast, integration testing failures arise from the complexities of module interactions[38], such as incompatible interfaces, data flow inconsistencies, and environmental mismatches between components.

For example, in the workflow execution "Test all JDKs on all OSes#977" [10], Gradle test failed due to unit test failure. During the execution of the workflow "Build and Test Scheduled On 3.1 #1596" [4], the integration test is the cause of the failure of the GHA workflow.

R150: *"Test flakes are probably most common."*

## 4.2 Failures Due to Workflow-Related Factors

Our manual analysis of sampled logs led to the identification of 60 GHA workflow failures related to workflow-specific factors, namely, the failures are not caused by the issues of the projects themselves.

These failures are further categorized into eight distinct types, related to either Environment/-Configuration issues or GitHub Issues, as can be seen in Figure 4. Similarly, the numbers in blue squares in Figure 4 indicate the frequency of each failure type observed during our analysis.

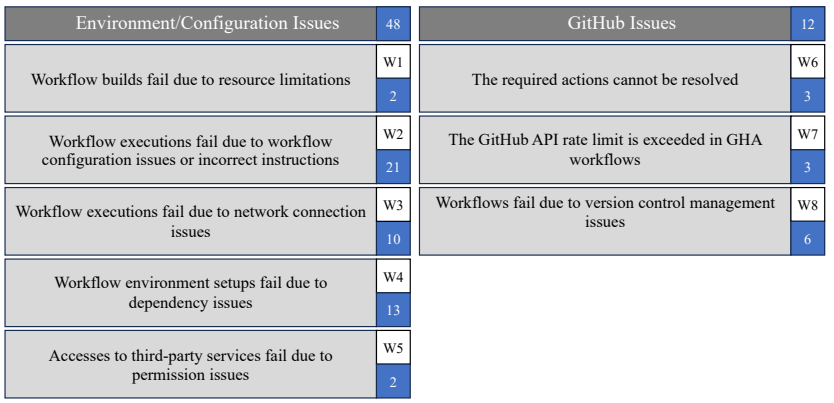



Fig. 4. Taxonomy of failures resulting from workflow-related factors.

*4.2.1 Environment/Configuration Issues.* These failures occur due to incorrect setups or limitations of the environment, or misconfigurations within the workflow.

**W1: Workflow builds fail due to resource limitations**. GHA workflows are executed on machines known as runners, which can be either GitHub-hosted or self-hosted. These runners come with different specifications such as operating systems, memory, and storage. Failures typically arise when the resources available are insufficient to meet the demands.

> ℹ For example, in the workflow execution "Push Images to Quay#265"[48], the issue encountered during the build process is related to insufficient disk space on the runner, specifically when attempting to build a Docker image.

> 👤 R1: "*Sometimes building for one specific platform (mostly Mac or Windows) fails. I think it is due to resource limitations for those platforms [...]*"

**W2: Workflow executions fail due to workflow configuration issues or incorrect instructions**. These failures may stem from misconfigured settings (e.g., wrong package manager used), incorrect or missing parameters in instructions, or syntax errors in the workflow steps.

> ℹ For example, in the workflow execution "Dependabot #704" [25], the error message indicates that the developer did not set the `GH_TOKEN` environment variable when using the GitHub CLI in the GHA workflow.

**W3: Workflow executions fail due to network connection issues**. Network-related issues typically arise from internet connectivity interruptions, server downtime, or issues with the underlying network infrastructure. Such failures can halt critical tasks, leading to delays and potential data loss.

> ℹ In the workflow execution "E2E Test #596" [5], *kubectl* installation failed due to a connection error (Connection reset by peer, errno 104).

**W4: Workflow environment setups fail due to dependency issues**. Failures in this category arise when essential components such as libraries, frameworks, or tools used in the workflow are missing, outdated, or incompatible with the system configuration.

> ℹ For example, in the workflow execution "OpenJDK GHA Sanity Checks #2007" [33], during the environment setup process, the system encountered an error indicating that the specified versions of certain packages were not found. In this case, the version "10.3.0-1ubuntu1~20.04" was not available for both *gcc-10* and *g++-10*.

**W5: Accesses to third-party services fail due to permission issues**. These failures typically indicate that the process lacks the necessary permissions or credentials to connect or interact with third-party services.

This can result in authentication failures or access denial, preventing the workflow from using the external services effectively. Resolving these issues typically involves reviewing and adjusting access permissions, ensuring that authentication credentials (such as API keys or tokens) are correctly configured within the application, and confirming that the user or service account has been granted appropriate permissions by the third-party service provider.

> ℹ For example, in the workflow execution "Synchronization code #263" [45], a permission issue arose when mirroring a repository from GitHub to another Git server.

> 👤 R129: "*Where workflows depend on organization-level secrets for credentials, and the organization follows best practices by not having secrets with unlimited validity, then those secrets will tend to expire over time. This can lead to large numbers of builds suddenly failing overnight when those secrets expire if the organization isn't proactively managing the rotation of those secrets.*"

*4.2.2 GitHub Issues.* In this category, we describe the failures associated with GitHub, especially those arising from its GHA workflow services. These issues can be further classified into three sub-categories.

**W6: The required actions cannot be resolved**. Actions are reusable extensions for executing repetitive instructions in GHA workflows. One notable feature of GHA compared to other CI/CD services is that developers can reuse actions created by others to simplify the workflow. In spite of the benefits, failures can arise when these actions cannot be resolved.

For instance, a repository may be inaccessible due to typos in the repository name or owner, deletion or privatization of the action repository, or the specified version (branch, tag, or commit SHA) not being available. We distinguish actions from other dependencies in W4 because actions are specific to GitHub, and their resolution occurs exclusively on the GitHub platform.

> ℹ For example, in the workflow execution "Java CI#396" [58], the action "codecov/codecov-action@v4" cannot be resolved, as the specified version "v4" could not be found.

**W7: The GitHub API rate limit is exceeded in GHA workflows**. The GitHub API offers two primary interfaces for developers: the REST API and the GraphQL API. These APIs allow developers to interact with the GitHub platform and manage projects. However, these APIs enforce rate limits to ensure fair usage and protect the GitHub infrastructure from abuse. When a workflow exceeds the rate limit, it can disrupt the execution of GHA workflows.

> ℹ For example, in workflow execution "Run Acceptance Tests from PR#629" [47], the GHA runner had surpassed the API rate limit when installing a utility CLI with Git. Therefore, the workflow execution was terminated.

**W8: Workflows fail due to version control management issues**. GHA workflows frequently handle tasks related to version control, a critical aspect of software development that ensures code integrity and collaboration among developers. However, failures can happen due to merge conflicts when multiple contributors make changes to the same part of a codebase, accidental creation of duplicate release tags, or other versioning problems (e.g., inconsistent tagging conventions or incorrect branch merges).

> ℹ For example, in the workflow execution "Sync project branch #165" [32], a merge conflict arose for the `pom.xml` file when merging the `origin/master` branch into the *gsoc-2023-project* branch.

### 4.3 CI-Dominated Failures in GitHub Actions Workflows

In addition to identifying the root causes of GitHub Actions workflow failures, our research also examined whether these failures occurred in the Continuous Integration (CI) or Continuous Deployment stage. We analyzed whether both phases exhibit similar failure patterns or face distinct challenges.

Our analysis of 375 failed workflow executions shows that most failures occur during the CI phase, with 370 of 375 cases originating at this stage. These failures are primarily caused by software test failures (P8), compilation errors (P1 to P4), and quality issues (P5 to P7). These issues can disrupt the CI process and carry over to later stages, ultimately affecting the entire development workflow.

In contrast, only 5 out of the 375 failures were observed in the Continuous Deployment phase. Among these, 3 errors occurred during the release versioning process, where software releases are assigned and managed version numbers. The remaining 2 failures were caused by insufficient deployment resources, such as inadequate memory. Although failures in Continuous Deployment

are relatively rare, they present distinct challenges related to release management and resource allocation, making them different from the issues typically encountered in CI.

The CI/CD pipeline consists of two interconnected phases that share some overlapping failure patterns but also exhibit unique challenges. Addressing root causes as early as possible is crucial for maintaining workflow stability. Enhancing CI practices through rigorous testing, build optimizations, and early error detection can prevent failures from propagating to the deployment stage. At the same time, improving resource management and implementing precise release control in Continuous Deployment is essential for mitigating deployment-specific risks.

### 4.4 Failure Distribution in Actions and Scripts

To gain deeper insights into the failure patterns within GHA workflows, we analyzed the root causes of 375 failure instances across actions and scripts. Actions, as predefined and reusable components, rely heavily on stable configurations and external integrations, making them more susceptible to workflow-related issues (W1–W8) such as misconfigurations, environmental inconsistencies, and network disruptions. In contrast, scripts are custom implementations tailored to specific projects. They interact directly with the repository's code, build tools, and testing frameworks, making them more prone to project-related issues (P1–P8) such as test failures, compilation errors, and dependency conflicts. This fundamental distinction is evident in their failure distributions: scripts account for the majority of failures (297 cases, 79.2%), reflecting their vulnerability to project-specific complexities, while actions contribute 78 failures (20.8%), with a higher proportion linked to workflow-related factors. These failures are further detailed in Table 3, which illustrates their distribution across 16 different types of failures.

Table 3. Distribution of Failures in Actions and Scripts

| Failure Type | Action Failures | Script Failures | Total |
|---|---|---|---|
| P1 | 4 | 50 | 54 |
| P2 | 7 | 12 | 19 |
| P3 | 2 | 31 | 33 |
| P4 | 6 | 39 | 45 |
| P5 | 4 | 22 | 26 |
| P6 | 3 | 12 | 15 |
| P7 | 0 | 3 | 3 |
| P8 | 8 | 112 | 120 |
| W1 | 1 | 1 | 2 |
| W2 | 12 | 9 | 21 |
| W3 | 8 | 2 | 10 |
| W4 | 10 | 3 | 13 |
| W5 | 2 | 0 | 2 |
| W6 | 3 | 0 | 3 |
| W7 | 3 | 0 | 3 |
| W8 | 5 | 1 | 6 |
| **Total** | **78** | **297** | **375** |

*4.4.1 Action Failures: Sensitivity to Workflow Issues.* Among the 78 action failures, 44 (56.4%) are workflow-related (W1–W8), while 34 (43.6%) are project-related (P1–P8). This distribution underscores actions' dependence on accurate workflow configurations and stable execution environments.

The most frequent workflow-related failure points include configuration errors (W2, 12 cases), environment setup problems (W4, 10 cases), and network interruptions (W3, 8 cases). Such failures typically stem from incorrect usage, missing dependencies, or networking issues intrinsic to the nature of actions as reusable extensions. In contrast, project-related failures, such as software test failures (P8, 8 cases), occur less frequently, reflecting the relative stability of actions when handling project-specific tasks. These findings suggest that enhancing workflow reliability through robust configuration validation and consistent environment management could substantially reduce action failures.

*4.4.2 Script Failures: Project Complexity as a Key Factor.* Scripts exhibit a markedly different failure pattern, with 297 recorded failures, of which 281 (94.6%) are project-related (P1–P8) and only 16 (5.4%) are workflow-related (W1–W8). This overwhelming bias toward project-related issues highlights the challenges inherent in custom scripts, which must accommodate diverse project-specific requirements and dependencies. The main sources of failure include software test failures (P8, 112 cases), and compilation errors. This pattern reflects the flexibility and inherent risk of tailoring scripts to meet specific project needs.

## 4.5 Prevalence of Identified Failure Causes in Our Taxonomy

To validate our taxonomy, we asked developers how often they encounter each type of failure. Given the responses, in this subsection, we analyze the frequency of failure causes and the difference between the survey responses and our manual analysis. We also discuss the impact of developer experience on the results.

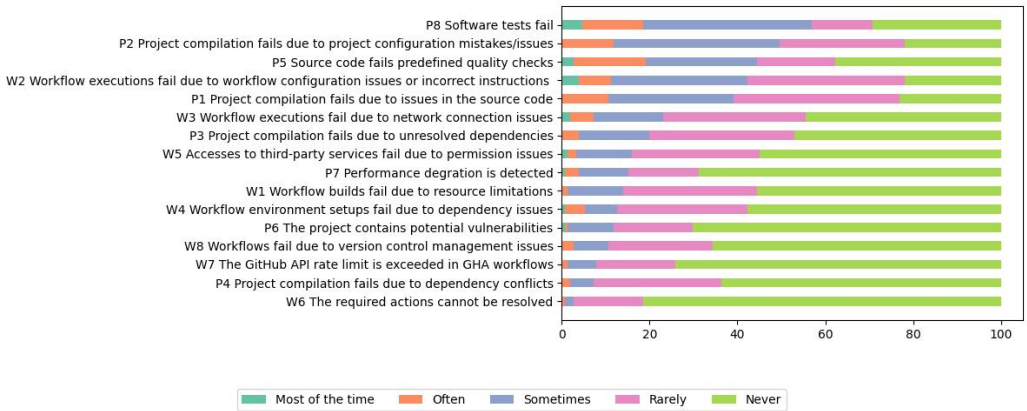



Fig. 5. Frequency at which the respondents encounter each failure in percentage

*4.5.1 Frequency of Failure Cause Occurrences in the Survey.* Figure 5 presents the occurrence frequency of each type of failure cause. The most frequently encountered workflow failure by developers is related to software testing (P8), with 20.5% of respondents reporting it as a frequent issue and 41.7% encountering it occasionally. Project build failures due to configuration mistakes/issues (P2) are also fairly common, with 60% of respondents experiencing this issue at least "sometimes". Additionally, code failing predefined quality checks (P5) and source code issues (P1) are notable as well, with around 50% of respondents reporting it as an issue they encounter at least occasionally. Among the top five most common causes of workflow execution failures, four of them (P8, P2, P5, P1) are project-related.

The only workflow-related failure in the top five is configuration issues or wrong instructions in the workflow (W2), which was also reported by around 50% of respondents as a failure they at least sometimes encounter. Unresolved dependencies (P3) and network connection issues (W3) are also relatively common, with about 25% of respondents experiencing this occasionally. All other remaining types of failure causes are rather rare for developers, as over 80% of survey respondents rarely or never experienced them. Interestingly, the least common failure cause is unresolvable actions (W6), indicating the overall reliability of third-party actions.

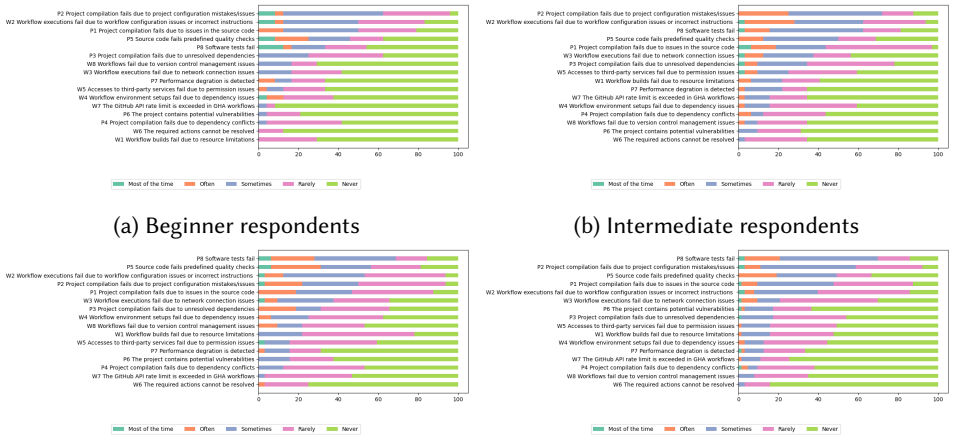
*4.5.2 Frequency Comparison Between Survey and Manual Analysis.* Our manual analysis of GHA workflow failures reveals that certain types of issues are particularly prevalent within CI/CD pipelines on GitHub. Notably, software test failures (P8) have emerged as the leading cause, accounting for approximately 32% of all recorded issues. This high percentage highlights the critical role that testing plays in the software development lifecycle, underscores the challenges developers face in ensuring the robustness of the code being tested, and suggests underlying issues related to test design and test data. Project compilation failures due to issues in the source code (P1) and dependency conflicts (P4) are also quite common, constituting about 14% and 12% of the total failures, respectively.

There are notable differences regarding the failure frequency ranks in manually analyzed issues and survey responses. For example, while project build failures due to dependency conflicts (P4) ranked as the third most common failure in the manual analysis, only 10% of survey respondents reported experiencing it more than rarely, which is among the three least common failures. To understand this disparity, we note that in the GHA environment, the impact of dependency conflicts is often confined to the build stage. These failures typically do not propagate to testing or deployment phases; if a CI/CD pipeline fails due to dependency issues, developers can address them promptly, preventing broader workflow disruptions. Moreover, dependency conflicts generally do not affect core business logic, functioning more as temporary build obstacles rather than systemic problems. Consequently, developers may not perceive P4 as a "major failure" and underreport its significance in the survey, focusing instead on failures with wider or more persistent impacts, such as test failures (P8).

Contrarily, regarding the failures related to access to third-party services due to permission issues (W5), several developers highlighted it as a significant concern in the survey, while only three instances of this issue were identified in our manual analysis. This could be attributed to different usages of third-party services between sampled projects and survey respondents, or more memorable experiences in fixing the issue.

*4.5.3 Impact of Developer Experience on the Results.* Developers' perceptions of failures can vary significantly based on their professional experience. To explore this, we analyze how survey responses differ across experience levels, utilizing frequency data from 151 respondents categorized into four groups: "beginner" (< 3 years, 24 respondents), "intermediate" (3-5 years, 32 respondents), "experienced" (6-10 years, 32 respondents), and "expert" (> 10 years, 63 respondents). Table 4 presents the frequency of each failure type encountered at least "sometimes" across these groups, providing a quantitative basis for our analysis. The failure frequency graphs for these groups are depicted in Figure 6.

Regardless of experience, the most common failure causes across all groups, as evidenced by the highest frequencies in Table 4, are project build failures due to configuration mistakes/issues (P2, ranging from 0.50 to 0.72), failure to pass predefined quality checks (P5, 0.46-0.56), software test failures (P8, 0.33-0.70), and workflow execution failures due to configuration issues or incorrect instructions (W2, 0.40-0.62). These consistent patterns suggest that certain failure types are pervasive challenges in GHA workflows, irrespective of expertise. Conversely, the least frequently




(a) Beginner respondents

(b) Intermediate respondents

(c) Experienced respondents

(d) Expert respondents

Fig. 6. Survey responses by experience level

Table 4. Frequency of Failure Types Across Experience Levels (more than rarely)

| Failure Type | Beginner | Intermediate | Experienced | Expert |
|---|---|---|---|---|
| P1 | 0.50 | 0.44 | 0.47 | 0.48 |
| P2 | 0.62 | 0.72 | 0.50 | 0.59 |
| P3 | 0.25 | 0.34 | 0.31 | 0.17 |
| P4 | 0.04 | 0.12 | 0.12 | 0.10 |
| P5 | 0.46 | 0.50 | 0.56 | 0.49 |
| P6 | 0.04 | 0.09 | 0.16 | 0.17 |
| P7 | 0.17 | 0.22 | 0.16 | 0.13 |
| P8 | 0.33 | 0.62 | 0.69 | 0.70 |
| W1 | 0.00 | 0.22 | 0.22 | 0.16 |
| W2 | 0.50 | 0.62 | 0.53 | 0.40 |
| W3 | 0.17 | 0.38 | 0.38 | 0.21 |
| W4 | 0.12 | 0.16 | 0.25 | 0.13 |
| W5 | 0.12 | 0.25 | 0.16 | 0.16 |
| W6 | 0.00 | 0.03 | 0.03 | 0.03 |
| W7 | 0.04 | 0.16 | 0.03 | 0.11 |
| W8 | 0.17 | 0.09 | 0.22 | 0.08 |

encountered failure across all groups of respondents is the inability to resolve the required action (W6).

Despite these similarities, notable differences emerge. Software test failures (P8) occur significantly less frequently among beginners (0.33) and intermediates (0.62) compared to experienced (0.69) and expert developers (0.70). To examine this association, we performed a chi-square test[20], yielding $\chi^2 = 10.65, p = 0.014 (< 0.05)$, confirming a significant relationship between experience level and the perception of P8 failures. This result suggests that software test failures tend to be more common among expert developers. A plausible explanation is that beginners and intermediates

often address issues earlier in development, before the formal testing phase, leveraging tools such as IDE static analysis and code reviews to catch errors like flawed logic or assertion mismatches [26, 52, 67]. This early intervention reduces the incidence of test failures at the CI stage. In contrast, experienced and expert developers typically engage in more complex large-scale projects involving architectural design, concurrency control, and sophisticated features. These challenges are less detectable by early-stage tools and more likely to manifest during integration or system testing [38], resulting in a higher rate of test failures (up to 0.70 for experts).

This finding suggests that tailored support and training may be necessary to help less experienced developers navigate complex challenges effectively. In addition, organizations could benefit from building collaborative environments where knowledge sharing between developers of varying experience levels is encouraged, thereby flattening the learning curve and improving overall workflow efficiency.

### 4.6 Challenges Encountered When Dealing with GHA Workflow Failures

In the survey, the respondents provided feedback on the major difficulties they encountered when dealing with GHA failures. These challenges can be categorized as follows.

**Difficulties in Local Testing.** Developers often struggle with testing workflows locally before pushing changes to remote repositories. Since GHA workflows cannot be fully replicated in a local environment, developers must rely on committing and pushing changes to verify their effectiveness. This process results in a cluttered commit history filled with intermediary commits and increases resource usage, as each push triggers a new build and test cycle on GitHub's infrastructure. The inability to simulate workflows locally slows the development cycle, leading to inefficiencies and increasing developer frustration.

> R136: "*No Workflow for local development, always against GitHub.* "

The situation is further complicated by the fact that sometimes developers need organization-level secrets to execute the workflow.

> R129: "*Debugging action failures can be awkward, while ways to run workflows locally exist, e.g. act, these aren't a realistic way to debug most organization workflows as those typically rely on organization secrets to which individual developers have no access.*"

**Complex Debugging.** Debugging workflows in GitHub Actions can be intricate and time-consuming. When a workflow fails, the debugging information provided is often limited, making it harder to identify the root cause.

> R35: "*Detailed configuration of the machine with the list of all the directories, network configuration, allowed/blocked network hosts, ssh connect ability, etc. are neither provided nor is it available.*"

Moreover, logs, essential for troubleshooting, are often very large. It is difficult to manually navigate the logs and locate the error.

> R8: "*Log presentation is often slow and hard to search and process manually.*"

Additionally, debugging complex workflows, particularly those involving multiple steps or external dependencies, is further complicated by the lack of SSH access to the running environment, impeding interactive diagnosis and resolution of issues.

> R23: "*The inability to ssh into the container by default to actually analyze the situation.*"

Another issue is the flaky tests. Test flakiness is a challenging problem in software testing. As tests are often executed in GHA workflows, it also inevitably becomes an issue for GHA workflow debugging.

> R150: "*Hard to track flakiness/test history of a particular test or GHA suite over time.*"

**YAML Configuration Issues.** The use of YAML as the configuration language for GHA poses its own set of challenges. The complex syntax of YAML can lead to maintenance difficulties, often resulting in syntax errors and configuration issues. Developers who are unfamiliar with GHA or YAML find the initial setup particularly daunting. Misconfigurations typically lead to multiple iterations of trial and error, consuming time and adding overhead, complicating the setup and maintenance of CI/CD pipelines.

> R5: "*YAML grammar is so tedious and hard to maintain.*"

**Resource Constraints.** GitHub Actions runners operate on machines with limited computing resources, which can cause concurrency issues and performance bottlenecks. For instance, unit tests that run smoothly in a local environment may encounter concurrency bugs on a shared, resource-constrained runner. This discrepancy between local and CI environments complicates the effort to ensure workflow reliability. Limited resources on GitHub Actions runners can also affect the execution speed and efficiency of builds and tests, exacerbating resource management challenges.

> R45: "*GitHub Actions runners use very busy machines with limited resources. For this reason, concurrency bugs in unit tests often appear in GHA, but are almost impossible to reproduce on a (mostly idle) developer desktop.*"

**External Services and Network Issues.** External services and network issues frequently affect the stability of GHA workflows. Downtime of external services or failures in dependency resolution can lead to unreliable builds and tests. Developers often have no control over these external services and it is also difficult to find a workaround.

These issues also result in intermittent failures that are difficult to diagnose and resolve, as they may not be consistently reproduced and can be influenced by external factors.

> R109: "*When I troubleshoot the failures, the reason usually is network issues, so I don't have any solutions to solve the problems, the only way is waiting for the dependencies maven repositories to recover.*"

> R147: *"Worker nodes getting shut down, unreliable infrastructure (network, etc.).*"

**Long Feedback Cycle.** The feedback cycle for addressing issues in GHA workflows is often prolonged. Even when developers make minor fixes, they have to commit the change and wait for the whole workflow to be executed. As fixing might not be done in one shot, it often leads to extended waiting periods between iterations. The slow pace of debugging and testing, coupled with the inability to run workflows locally, makes resolving issues particularly cumbersome. This extended feedback loop hinders rapid development and continuous improvement, slowing down overall workflow optimization.

> R136: "*Long feedback loop and feels like tinkering rather than informed engineering.*"

> R51: "*Troubleshooting is tedious, especially in the beginning when setting up the workflow. It is a lot of trial-and-error that requires a complete commit-push-compile cycle every time you make the smallest change in the configuration files. It may take half a dozen such cycles before you get it right, which takes time.*"

**Documentation Issues.** Although GitHub Actions is well-documented, navigating the extensive documentation to find relevant information can be challenging. The volume of documentation can make it difficult to locate specific details or examples needed for particular issues or configurations. For instance, developers may struggle to find appropriate guidance or example configurations for tasks such as code signing or automated testing of pull requests that involve using secrets.

> R102: "*Finding the right documentation. The documentation for GitHub Actions is very good, but there is a lot of it.*"

**Synchronization of workflow updates.** Another challenge occurs when developers are maintaining multiple projects or multiple versions with similar workflow configurations. The patch to fix an issue in a workflow file will not be automatically propagated to other workflows with the same issue. When a lot of workflows need to be updated, it could be time-consuming and easy to ignore.

> R149: "*As I have many projects following the same template (same gradle plugins, versions, etc) not all of them are actively worked on, or have multiple branches that don't get changes as they are for lts/legacy versions, so when a failure like this happens it may go unnoticed for a long time and require me to update several places to resolve across all of my projects*".

### 4.7 Support Needed for Better Handling GHA Workflow Failures.

Developers also provide opinions on what kind of support they need to address the GHA workflow failures. These specific requirements indicate the potential research directions for researchers.

**Local workflow execution.** Most developers mentioned that they would like to execute the workflow locally. Running workflows locally helps identify and fix issues early in the development cycle, thereby enhancing the reliability and efficiency of CI/CD pipelines. Although there is an existing tool named Act[6], many developers remain unaware of it. Act mimics the GitHub Actions environment on local machines, enabling developers to test and debug workflows before pushing them to the repository. Despite its power, Act is not a perfect tool. It does not fully replicate all features and environments of GitHub Actions, such as the support matrix and Docker context, which can lead to workflows behaving differently locally compared to the GitHub CI/CD environment. Additionally, Act may not accurately reflect the actual behavior of workflows that rely on confidential or proprietary organizational data, making it unsuitable for many enterprise-level scenarios. Many developers are still hoping that GitHub will officially provide such a tool that can test their workflow locally and seamlessly.

**Workflow testing tools.** Other developers have asked for better tools to test workflow files. Currently, several tools are available to assist developers in checking the syntax and format of YAML files, such as YAML Lint[7] and PyYAML[8]. These tools perform static analysis, ensuring that YAML files are free from syntax errors and adhere to best practices. However, these checks are rather basic and cannot handle complex issues in workflow files. Indeed, solving workflow issues would often require executing the workflow itself, which further highlights the necessity of local workflow execution tools. More testing approaches should be built upon these tools to reduce the GHA workflow failures on the GitHub server.

**Interactive interface for execution result management.** Developers who complained about lengthy logs also asked for better ways to inspect the logs. Currently, when the logs are too long, they will be truncated on the GitHub interface. While developers can download the log files, a bunch of txt files make it very difficult to identify the cause of failures. An interactive log interface that integrates advanced log analysis techniques would significantly improve the process, helping developers troubleshoot more effectively and efficiently.

## 5 DISCUSSIONS

In this section, we discuss the implications of our findings.

[6]act, https://github.com/nektos/act
[7]yamllint, https://www.yamllint.com
[8]PyYAML, https://pyyaml.org

### 5.1 Comparative Analysis of Failure Causes in GHA and Older CI/CD Tools

This subsection compares failure causes between GHA and older CI/CD tools, such as Jenkins and Travis CI. By highlighting similarities and differences, we illustrate how GHA's architecture influences its failure patterns, supported by empirical data and prior research.

*5.1.1 Similarities in Failure Causes.* Despite differences in CI/CD tools, certain failure types persist across platforms such as GHA, Travis CI, and Jenkins, particularly in project-related and resource-related challenges. Software test failures are the most frequent issue in GHA, accounting for 32% (120 out of 375 cases) of failures and reported by 62.2% of survey respondents. These findings align with prior studies by Rausch et al. [49] and Beller et al. [6], which identified testing challenges as a prevalent issue in Travis CI and Jenkins.

Compilation failures, including source code errors (14.4%), configuration mistakes, unresolved dependencies, and dependency conflicts (12%)—are another common issue, mirroring Zhang et al.'s [66] observations of compiler errors in Travis CI. Similarly, quality check failures and configuration issues affected at least 60% of GHA users at some point, resembling the misconfiguration problems reported in Travis CI by Pinto et al. [46].

Additionally, resource limitations (W1), where workflows fail due to insufficient disk space or memory on runners, are not exclusive to GHA. Travis CI also encounters such constraints under heavy workloads [21]. These recurring failure patterns suggest that many CI/CD challenges stem from fundamental software development and execution issues rather than tool-specific shortcomings.

*5.1.2 Distinct Failure Causes in GHA Compared to Older CI/CD Tools.* While GHA shares some failure patterns with older CI/CD tools, GHA also introduces unique failure modes due to its deep integration with GitHub and reliance on reusable marketplace actions, particularly in workflow-related failures. One notable issue is the inability to resolve required actions (W6), where workflows fail due to typos, repository inaccessibility, or versioning issues in referenced actions [16]. Unlike GHA, Jenkins primarily relies on custom scripts and plugins, which, although sometimes unavailable, do not depend on an external action resolution mechanism. Similarly, Travis CI defines build steps through YAML configurations but lacks a centralized action marketplace, making it less susceptible to such failures.

Another distinct challenge is exceeding GitHub API rate limits (W7), which can disrupt workflow execution due to GHA's heavy reliance on GitHub's API at multiple stages. In contrast, Jenkins, which is often self-hosted, does not inherently depend on the GitHub API. Meanwhile, despite interacting with GitHub for repository access and status updates, Travis CI operates on independent infrastructure, making it less vulnerable to this limitation.

These distinct failure modes highlight GHA's unique dependencies and configuration challenges compared to older CI/CD tools, underscoring how its tight GitHub integration introduces novel failure risks that are less pronounced in Jenkins and Travis CI.

### 5.2 Leveraging GHA Failure Taxonomy for CI/CD Enhancement

Our taxonomy and survey results offer actionable insights for stakeholders to improve CI/CD pipelines in GHA. By systematically addressing the failure patterns identified in the taxonomy and integrating developer feedback from the survey, these insights pave the way for targeted improvements across the CI/CD ecosystem. The taxonomy, built from an analysis of 375 failed executions and validated by 151 developers, classifies failures into 16 types. Coupled with survey data on failure frequency and developer experiences, it shifts the focus from reactive fixes to proactive, data-driven strategies. The following sections elaborate on how each stakeholder group

can apply these findings, with specific examples and recommendations to illustrate their practical utility.

**Diagnostic Guidance for Developers.** Developers can leverage the taxonomy and survey insights to diagnose and mitigate failures in GHA workflows more effectively. By correlating failures with specific taxonomy categories, developers can classify issues, identify recurring patterns, and implement proactive solutions—going beyond the limitations of traditional error logs. While error logs provide detailed, incident-specific insights, such as test failures or permission errors, the taxonomy uncovers broader systemic implications, allowing teams to address root causes rather than merely resolving individual failures as they arise. A key insight from the taxonomy is the high frequency of software test failures (P8), which account for 32% of all reported cases. Given the prevalence of P8 failures, enhancing test reliability should take precedence over addressing less frequent issues. Such strategic prioritization ensures that teams allocate resources effectively, focusing on the most critical pain points in their CI/CD pipelines.

By leveraging the taxonomy's structured classification and survey-driven insights, developers can shift from reactive debugging to proactive problem-solving. Rather than repeatedly fixing the same failures in isolation, teams can address failure trends, implement long-term solutions, and strengthen the overall resilience of their CI/CD workflows. For example, organizations can establish a monthly audit script to check for outdated actions (W6) and prompt updates, preventing workflow disruptions caused by deprecated functionalities. Similarly, enforcing secret rotation policies (W5) can help prevent unexpected permission-related failures, ensuring continuous pipeline stability.

**Tool Development Opportunities.** The taxonomy and survey highlight key areas for tool innovation to enhance CI/CD workflows. One promising opportunity is addressing dependency conflicts (P4). A survey participant(R15) expressed the need for a tool that could analyze dependency trees across workflows, recommend version alignment, and automatically update .github/workflows files. For example, detecting incompatible Node.js versions between workflows could prompt a recommendation for a unified version, reducing build failures and simplifying dependency management.

Resource optimization also offers potential, especially for resource limits (W1). Predictive tools could use historical workflow data to allocate resources dynamically, preventing runtime failures. A resource allocator might forecast peak demands for a matrix build and pre-provision runner capacity, ensuring seamless execution. Such innovations bridge developer needs with practical solutions, setting the stage for platform-level enhancements.

**Platform-Level Improvements.** GitHub itself can elevate the CI/CD experience by addressing systemic issues revealed in the taxonomy and survey. To alleviate concerns about resource limitations, GitHub could offer real-time CPU/memory usage dashboards. A job-specific resource consumption view would help developers tweak matrix configurations, avoiding failures and boosting performance.

For local testing challenges, some survey participants (e.g., R59, R62, R70) suggested that a native testing environment mirroring the cloud setup could catch issues like unresolved dependencies before committing. This pre-pipeline validation would save time and resources, aligning with the goal of a more robust CI/CD ecosystem.

## 6 THREATS TO VALIDITY

*Threats to construct validity* concern the relation between theory and observation. Our study sampled our data with a 95% confidence interval with a ±5% margin of error. While this is a commonly used sampling approach in software engineering empirical studies, the results might still be impacted by the chosen interval values and the randomness of data sampling.

Besides, in this study, we focused on Java projects in general, without considering the impact of the project types (e.g., web applications and Android apps). We acknowledge that different technology stacks may exhibit varying patterns that could diverge from our expected results.

Another potential threat to construct validity lies in the representativeness of the failures we analyzed. Our taxonomy is derived from 375 failed GHA workflow executions, selected randomly from a pool of 15,355 valid logs across 260 Java projects. While this sample size is statistically sufficient, it may not fully capture the diversity of failure scenarios encountered in practice, particularly rare or context-specific issues that occur infrequently or in niche use cases. For example, failures tied to specific GHA features (e.g., matrix builds or self-hosted runners) might be underrepresented. This limitation may lead to an overrepresentation of failures that are easier to detect and categorize, such as software test failures, while potentially underestimating complex, multi-faceted issues that require deeper contextual understanding. To mitigate this, we supplemented our manual analysis with a survey of 151 developers, which broadens the perspective.

In our survey to gather developer feedback, we adopted the 5-level Likert scale to assess the prevalence of each type of failure. While this is also a widely used approach in surveys, survey participants might interpret these options differently, potentially introducing biases. Given the fact that different software systems have very different sizes and GHA workflow usage practices, it is impractical to quantify the frequencies in the survey. The goal of our study is not to understand how many times each type of failure occurs, instead, we aim to capture the relative prevalence of different failures. Therefore, we believe the current setting of the survey can well serve the purpose of this study.

*Threats to internal validity* concern extra factors we did not consider that could impact the results. Given the nature of our study, one major factor that might introduce bias to our study is the subjectivity of annotators. To mitigate this issue, each log was independently examined by at least two authors with a Cohen's Kappa coefficient of 0.708, indicating considerable agreement [41]. Conflicts were also thoroughly discussed among the authors until agreement was reached.

During the annotation process, the annotators were encouraged to reuse the previously created labels. One possible consequence is that the annotators might choose a label that is relevant but not 100% precise. To avoid this issue, we asked the annotators to include more details in the root cause. This guarantees the accuracy of tags, while sacrificing the purpose of label reuse to a certain extent.

*Threats to external validity* concern the generalizability of the results obtained in this study. In this study, we only investigated Java projects on GitHub. While Java is one of the most widely used programming languages, it is unclear whether the results can be fully applicable to projects developed in other programming languages.

Moreover, the studied projects are all open-source projects due to the necessity to access workflow execution logs. It still needs to be examined whether the closed-source software shares similar patterns. Future work can replicate our study and incorporate data from more diverse sources, such that our findings can be validated in different languages and project types.

## 7 CONCLUSIONS

GHA workflow failures occur frequently in practice, hindering developer productivity and wasting computational resources. In this study, we provide a comprehensive analysis of GHA workflow failures, offering valuable insights into the types and root causes of these failures in the context of GHA. By examining 375 failed GHA workflow executions from 260 open-source Java projects, we established a taxonomy containing 16 distinct types of failure causes. These categories, accompanied by real-world examples, may help developers diagnose and address specific issues in their GHA workflows, ultimately improving the reliability and efficiency of software development

processes. Our results were further validated through a survey of active GHA users, ensuring the relevance and comprehensiveness of the taxonomy proposed.

Our results also reveal several challenges developers encounter when dealing with GHA workflow failures, such as local testing of workflows and resource constraints. Addressing these challenges can further improve the maintainability and reliability of GitHub CI pipelines. Using the results obtained in this study as a foundational stepping stone, our future work lies in developing automatic approaches to tackle these challenges and advancing modern practices of CI/CD usage in software development projects.

## ACKNOWLEDGMENT

This work is partly supported by the National Natural Science Foundation of China (Grant Nos. 62302347 and U2436205).

## REFERENCES

[1] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. 2017. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439* (2017).
[2] Amazon Ion. 2023. Ion Java GitHub Actions Run #6829246558, Job #18575023797. https://github.com/amazon-ion/ion-java/actions/runs/6829246558/job/18575023797 Last accessed on 08-06-2024.
[3] Andrey Novikov. 2023. Trekarta GitHub Actions Run #6631013238, Job #18013616500. https://github.com/andreynovikov/trekarta/actions/runs/6631013238/job/18013616500 Last accessed on 08-06-2024.
[4] Apache. 2023. Dubbo GitHub Actions Run #6835753665, Job #18589976238. https://github.com/apache/dubbo/actions/runs/6835753665/job/18589976238 Last accessed on 08-06-2024.
[5] aws-containers. 2023. aws-containers/retail-store-sample-app GitHub Actions Run #6723607295, Job #18274043718. https://github.com/aws-containers/retail-store-sample-app/actions/runs/6723607295/job/18274043718 Last accessed on 08-06-2024.
[6] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *2017 IEEE/ACM 14th International conference on mining software repositories (MSR)*. IEEE, 356–367.
[7] Barry W Boehm, John R Brown, and Myron Lipow. 1976. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*. 592–605.
[8] Islem Bouzenia and Michael Pradel. 2024. Resource Usage and Optimization Opportunities in Workflows of GitHub Actions. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.
[9] Guillaume Cardoen, Tom Mens, and Alexandre Decan. 2024. A dataset of GitHub Actions workflow histories. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 677–681.
[10] Casid. 2023. JTE GitHub Actions Run #6749461418, Job #18349849068. https://github.com/casid/jte/actions/runs/6749461418/job/18349849068 Last accessed on 08-06-2024.
[11] CC-Tweaked. 2023. CC-Tweaked GitHub Actions Run #6725853717, Job #18281028796. https://github.com/cc-tweaked/CC-Tweaked/actions/runs/6725853717/job/18281028796 Last accessed on 08-06-2024.
[12] An Ran Chen, Tse-Hsun Peter Chen, and Shaowei Wang. 2022. T-evos: A large-scale longitudinal study on ci test execution and failure. *IEEE Transactions on Software Engineering* (2022).
[13] Crazy Marvin. 2023. Morse GitHub Actions Run #6672595913, Job #18136820559. https://github.com/crazy-marvin/morse/actions/runs/6672595913/job/18136820559 Last accessed on 08-06-2024.
[14] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling projects in github for MSR studies. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 560–564.
[15] Wayne W Daniel and Chad L Cross. 2018. *Biostatistics: a foundation for analysis in the health sciences*. Wiley.
[16] Alexandre Decan, Tom Mens, and Hassan Onsori Delicheh. 2023. On the outdatedness of workflows in the GitHub Actions ecosystem. *Journal of Systems and Software* 206 (2023), 111827.
[17] Zishuo Ding, Jinfu Chen, and Weiyi Shang. 2020. Towards the use of the readily available tests from the release pipeline as performance tests: Are we there yet?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1435–1446.
[18] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*. Pearson Education.
[19] Martin Fowler and Matthew Foemmel. 2006. Continuous integration.

[20] Todd Michael Franke, Timothy Ho, and Christina A Christie. 2012. The chi-square test: Often used and more often misinterpreted. *American journal of evaluation* 33, 3 (2012), 448–458.

[21] Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. 2018. Noise and heterogeneity in historical build data: an empirical study of travis ci. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 87–97.

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

[23] Inc. GitHub. 2024. Storing and sharing data from a workflow. https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/storing-and-sharing-data-from-a-workflow Accessed: 2024-08-06.

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

[25] Google Maps Samples. 2023. Android Samples GitHub Actions Run #6850982659, Job #18626307968. https://github.com/googlemaps-samples/android-samples/actions/runs/6850982659/job/18626307968 Last accessed on 08-06-2024.

[26] Paul Hamill. 2004. *Unit test frameworks: tools for high-quality software development.* " O'Reilly Media, Inc.".

[27] Avijit Hazra. 2017. Using the confidence interval confidently. *Journal of thoracic disease* 9, 10 (2017), 4125.

[28] Jez Humble and David Farley. 2010. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education.

[29] Pamela Hunter. 2016. Margin of error and confidence levels made simple. *Retrieved February 25th* (2016).

[30] Hyperledger Fabric. 2023. Fabric Gateway Java GitHub Actions Run #6319347700, Job #17160112108. https://github.com/hyperledger/fabric-gateway-java/actions/runs/6319347700/job/17160112108 Last accessed on 08-06-2024.

[31] Idempiere. 2023. Idempiere GitHub Actions Run #6811617984, Job #18522322951. https://github.com/idempiere/idempiere/actions/runs/6811617984/job/18522322951 Last accessed on 08-06-2024.

[32] Jenkins CI. 2023. GitLab Plugin GitHub Actions Run #6739653239, Job #18321512657. https://github.com/jenkinsci/Gitlab-plugin/actions/runs/6739653239/job/18321512657 Last accessed on 08-06-2024.

[33] JetBrains. 2023. JetBrains Runtime GitHub Actions Run #6827200507, Job #18568801104. https://github.com/jetbrains/jetbrainsruntime/actions/runs/6827200507/job/18568801104 Last accessed on 08-06-2024.

[34] Stephen H Kan. 2003. *Metrics and Models in Software Quality Engineering*. Addison-Wesley.

[35] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. 2014. Why do automated builds break? an empirical study. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 41–50.

[36] Timothy Kinsman, Mairieli Wessel, Marco A Gerosa, and Christoph Treude. 2021. How do software developers use github actions to automate their workflows?. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 420–431.

[37] Igibek Koishybayev, Aleksandr Nahapetyan, Raima Zachariah, Siddharth Muralee, Bradley Reaves, Alexandros Kapravelos, and Aravind Machiry. 2022. Characterizing the security of github {CI} workflows. In *31st USENIX Security Symposium (USENIX Security 22)*. 2747–2763.

[38] Hareton KN Leung and Lee White. 1990. A study of integration testing and software regression at the integration level. In *Proceedings. Conference on Software Maintenance 1990*. IEEE, 290–301.

[39] Yiling Lou, Zhenpeng Chen, Yanbin Cao, Dan Hao, and Lu Zhang. 2020. Understanding build issue resolution in practice: symptoms and fix patterns. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 617–628.

[40] Antonio Mastropaolo, Fiorella Zampetti, Gabriele Bavota, and Massimiliano Di Penta. 2024. Toward Automatically Completing GitHub Workflows. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[41] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.

[42] Olivier Nourry, Yutaro Kashiwa, Bin Lin, Gabriele Bavota, Michele Lanza, and Yasutaka Kamei. 2023. The human side of fuzzing: Challenges faced by developers during fuzzing activities. *ACM Transactions on Software Engineering and Methodology* 33, 1 (2023), 1–26.

[43] Adam Oliner, Archana Ganapathi, and Wei Xu. 2012. Advances and challenges in log analysis. *Commun. ACM* 55, 2 (2012), 55–61.

[44] Openmf. 2023. Openmf GitHub Actions Run #6767425387, Job #18390040404. https://github.com/openmf/mobile-wallet/actions/runs/6767425387/job/18390040404 Last accessed on 08-06-2024.

[45] Pig Mesh. 2023. Pig Mesh GitHub Actions Run #6622836485, Job #17988915393. https://github.com/pig-mesh/pig/actions/runs/6622836485/job/17988915393 Last accessed on 08-06-2024.

[46] Gustavo Pinto, Fernando Castor, Rodrigo Bonifacio, and Marcel Rebouças. 2018. Work practices and challenges in continuous integration: A survey with Travis CI users. *Software: Practice and Experience* 48, 12 (2018), 2223–2236.

[47] Pulumi. 2023. Pulumi EKS GitHub Actions Run #6821578644, Job #18552321847. https://github.com/pulumi/pulumi-eks/actions/runs/6821578644/job/18552321847 Last accessed on 08-06-2024.

[48] Quarkus IO. 2023. Quarkus Images GitHub Actions Run #6520792390, Job #17708710967. https://github.com/quarkusio/quarkus-images/actions/runs/6520792390/job/17708710967 Last accessed on 08-06-2024.
[49] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 345–355.
[50] G David Ripley and Frederick C Druseikis. 1978. A statistical analysis of syntax errors. *Computer Languages* 3, 4 (1978), 227–240.
[51] 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.
[52] Per Runeson. 2006. A survey of unit testing practices. *IEEE software* 23, 4 (2006), 22–29.
[53] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. 2020. Predicting continuous integration build failures using evolutionary search. *Information and Software Technology* 128 (2020), 106392.
[54] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. 2022. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering* 29, 1 (2022), 21.
[55] Sk Golam Saroar and Maleknaz Nayebi. 2023. Developers' perception of GitHub Actions: A survey analysis. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. 121–130.
[56] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE access* 5 (2017), 3909–3943.
[57] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.
[58] Spotify. 2023. Missing Link GitHub Actions Run #6296482729, Job #17091557253. https://github.com/spotify/missinglink/actions/runs/6296482729/job/17091557253 Last accessed on 08-06-2024.
[59] Daniel Ståhl and Jan Bosch. 2013. Experienced benefits of continuous integration in industry software product development: A case study. In *The 12th iasted international conference on software engineering,(innsbruck, austria, 2013)*. 736–743.
[60] Artur Strzelecki, Karina Cicha, Mariia Rizun, and Paulina Rutecka. 2024. Acceptance and use of ChatGPT in the academic community. *Education and Information Technologies* (2024), 1–26.
[61] TIOBE Software. 2023. TIOBE Index - The Software Quality Company. https://www.tiobe.com/tiobe-index/ Accessed: 2023-11-23.
[62] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. 2017. A tale of CI build failures: An open source and a financial organization perspective. In *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 183–193.
[63] Jing Xia and Yanhui Li. 2017. Could we predict the result of a continuous integration build? An empirical study. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 311–315.
[64] Zalando. 2023. Logbook GitHub Actions Run #6837892949, Job #18594402544. https://github.com/zalando/logbook/actions/runs/6837892949/job/18594402544 Last accessed on 08-06-2024.
[65] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun Chen. 2019. Studying the characteristics of logging practices in mobile apps: a case study on f-droid. *Empirical Software Engineering* 24 (2019), 3394–3434.
[66] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. 2019. A large-scale empirical study of compiler errors in continuous integration. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 176–187.
[67] Hong Zhu, Patrick AV Hall, and John HR May. 1997. Software unit test coverage and adequacy. *Acm computing surveys (csur)* 29, 4 (1997), 366–427.
[68] Hao-Nan Zhu, Kevin Z Guan, Robert M Furth, and Cindy Rubio-González. 2023. Actionsremaker: Reproducing github actions. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 11–15.
[69] Celal Ziftci and Jim Reardon. 2017. Who broke the build? Automatically identifying changes that induce test failures in continuous integration at Google scale. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 113–122.