Beyond the YAML File: Understanding Real-World GitHub Actions Workflow Adoption

Ali Khatami
s.khatami@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Carolin Brandt
c.e.brandt@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Andy Zaidman
a.e.zaidman@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Abstract

Continuous Integration and Continuous Deployment (CI/CD) have become fundamental to modern software development, with GitHub Actions (GHA) emerging as a dominant automation platform. In this study, we analyze real-world execution records of GHA, examining how developers react to workflow failures, how these workflows are utilized by projects, and how these aspects relate to project characteristics. We quantitatively analyze 258,300 workflow run records from 952 repositories and perform an in-depth qualitative analysis of 21 selected, diverse GitHub repositories to understand how maintainers and contributors interact with workflow results. We identify three distinct failure response patterns, observe that higher usage intensity of GHA workflows correlates with lower failure rates, and uncover a configuration-usage gap where the presence of configuration files masks disabled or unused workflows. Moreover, our qualitative analysis of relationships between project characteristics and utilization patterns yields five hypotheses for future validation.

CCS Concepts

• **Software and its engineering → Designing software**; **Development frameworks and environments**.

Keywords

GitHub Actions, Workflow Runs, CI/CD

ACM Reference Format:
Ali Khatami, Carolin Brandt, and Andy Zaidman. 2018. Beyond the YAML File: Understanding Real-World GitHub Actions Workflow Adoption. In *Proceedings of The 30th International Conference on Evaluation and Assessment in Software Engineering (EASE 2026).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Continuous Integration and Continuous Deployment (CI/CD) have become pillars of modern software development, enabling teams to automate testing, building, and deployment processes [4, 5, 24, 16, 17, 18, 31, 36]. GitHub Actions (GHA), introduced in 2019, has rapidly emerged as one of the most widely adopted CI/CD platforms, offering developers a powerful yet accessible framework for workflow automation directly integrated with their source code repositories [12, 42, 28]. While the potential benefits of workflow automation are well-documented [24, 40], realization of these benefits depends on how development teams utilize tools in practice [29].

Existing studies [8, 12, 41] analyze static workflow configuration (YAML) files defining automation steps, taking the presence of configuration files as a proxy for usage. However, studies on static analysis tools revealed a significant gap between configured tools and their actual usage [3, 25]. Many projects maintain configuration files for tools that are not or rarely used, or whose results are consistently ignored by developers [7]. This disconnect suggests that configuration files alone provide an incomplete view of how automation technologies impact software development processes.

In this study, we instead examine *workflow run data*: the execution records of GHA workflows. This includes the events that trigger runs and run outcomes. We qualitatively analyze the responses of developers to failures through subsequent commits and pull request (PR) discussions to develop a richer understanding of GHA's role in modern software development processes. We address the following research questions:

RQ1 What are the observed GitHub Actions utilization patterns?

Following a quantitative approach for RQ1, we examine utilization intensity (number of runs and failure rates) and trigger patterns (events causing workflow executions). This reveals how frequently failures occur and whether intensive workflow adoption correlates with failure rates.

RQ2 How do developers react to GitHub Actions workflow failures?

To understand the effect of GHA adoption on development processes, we first focus on failure points in pull requests and main branches, as they enable us to analyze subsequent developer actions that address these failures.

RQ3 What relationships exist between project characteristics and GitHub Actions utilization patterns?

For RQ3 we examine relationships between project characteristics and the patterns identified in RQ2 and RQ1 through a qualitative analysis of 21 repositories. Given the complexity beyond simple categorical mappings, we propose hypotheses for future validation rather than definitive conclusions.

To answer these questions, we quantitatively analyze run records from 765 repositories based on a dataset of 952 GitHub projects [8].

EASE 2026, Glasgow, Scotland, United Kingdom

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

Then we systematically select 21 repositories for a qualitative analysis based on three key dimensions: popularity, workflow run frequency, and failure rates. We conduct in-depth analysis of each repository, examining workflow run records, developer interactions in PRs, associations between workflow runs and commits (both in PRs and the main branch), and the contextual characteristics of projects. Previous studies established that quality assurance (QA) practices vary considerably across different development contexts [27]. The adoption and effectiveness of these practices are influenced by numerous factors including team size, technology stack, and development methodology. GHA workflows, as a mechanism for automating QA activities, likely exhibit similar contextual dependencies.

By examining how development teams interact with GitHub Actions in practice, our research **contributes a more nuanced understanding of GitHub Actions workflow adoption** than configuration analysis alone could provide. We identify patterns in how developers respond to workflow failures, recognizing common workflow utilization patterns, and examine relationships between these patterns and project characteristics. These findings can be the basis for future studies of GHA practices, grounding them in rich observational data.

Our paper is structured as follows: in Section 2 we review GitHub Actions studies. We then describe our methodology for our mixed-methods analysis in Section 3. In Sections 4 through 6 we present our findings regarding workflow utilization patterns across different project contexts (RQ1), developer responses to workflow failures (RQ2), and we explore relationships between these patterns and project characteristics (RQ3). Section 7 presents a discussion of the implications of our study. We conclude our paper with Section 8.

2 Background

GitHub Actions (GHA), introduced in 2019, has rapidly become the dominant Continuous Integration and Continuous Delivery (CI/CD) automation tool within the GitHub ecosystem, largely due to its deep integration and the provision of free resources to open source projects [14, 21, 35]. Quantitative studies of GHA configuration files demonstrated widespread and growing adoption: 22% of popular projects adopted GHA according to early studies [9], increasing to 37% for popular GitHub projects and up to 57% for repositories utilizing the most popular programming languages [1]. Previous research explored various facets of GHA adoption:

2.1 Impact on Development Practices

Research on the impact of GHA shows that its adoption generally correlates with faster resolution latency for pull requests (PRs) and issues, and increased commit frequency [9]. However, other studies report a more nuanced picture, noting an increase in rejected PRs and fewer commits in merged PRs [41, 30], with accepted PRs receiving more discussion comments and taking longer to merge after adoption [41].

2.2 Ecosystem Structure and Evolution

Studies describe GHA workflows and reusable components (Actions) as a complex, rapidly evolving ecosystem [12, 11]. Workflows, defined by jobs and steps, undergo continuous modification [12] including fixing YAML syntax errors, debugging, and updating instructions, leading researchers develop specialized tools for tracking commit changes of workflow configurations [34, 38]. Although action reuse is widespread [12, 11], most workflows reference outdated releases, typically lagging at least seven months behind the latest version [11]. Moreover, analyses estimate an average annual cost of $504 for GHA adoption of paid-tier repositories, while optimizations in resource consumption are underutilized [8].

2.3 Security and Vulnerability Management

A major focus of studies has been the security risks in the GHA workflow configurations [1, 14, 6, 15, 32]. 99.8% of workflows are over-privileged with default read-write access, and 23.7% are exploitable for arbitrary code execution via PRs [32]. Vulnerable third-party actions are common [32], and security recommendations such as using commit hashes are followed by fewer than 3% of repositories [11]. Security tools like CodeQL remain underused; only 13.5% of top repositories enabling CodeQL actually use it [1]. Developers also cite security and complexity concerns when selecting actions [37].

Most empirical research statically analyzes workflow configuration (YAML) files found in the *.github/workflows* directory, using their presence as a proxy for adoption and usage [8, 12, 41]. Only Bouzenia and Pradel [8] focus on resource optimization of GHA workflows. The novelty of this paper is to analyze workflow run data, i.e., the execution records of GHA workflow runs including their trigger events and outcomes with qualitative analysis of developers' reactions to them. This approach is necessary to understand actual workflow adoption beyond configuration presence.

3 Methodology

We conducted an exploratory analysis of GitHub Actions (GHA) workflow adoption through a mixed methods analysis of workflow runs. This section explains our data collection process, repository sampling strategy, and the manual analysis approach used to answer our research questions. An overview is depicted in Figure 1.

Our study focuses on analyzing execution data from GHA workflows and related contextual information of repositories. We examine the recorded workflow runs[1] from the GitHub API, whether these runs pass or fail, how they were triggered, their associated pull requests (PRs) or commits, and how developers react to these outcomes. We define *workflow run data* as the execution history of GHA workflows, including timestamps, trigger events[2], GitHub page URLs, status and conclusion, associated commits, PRs, users, and branches. We examine these execution records in combination with observable developer reactions on PR and branch web pages.

3.1 Data Collection

Our goal is to identify GitHub repositories that actively use GHA and, thus, have workflow run data available for analysis. This presents two challenges: (1) finding a representative sample of

[1]"Workflow Run" is the exact terminology used by GitHub, more details of these records are available at https://docs.github.com/en/rest/actions/workflow-runs?apiVersion=2022-11-28#get-a-workflow-run

[2]Events that trigger workflows: https://docs.github.com/en/actions/reference/workflows-and-actions/events-that-trigger-workflows#about-events-that-trigger-workflows

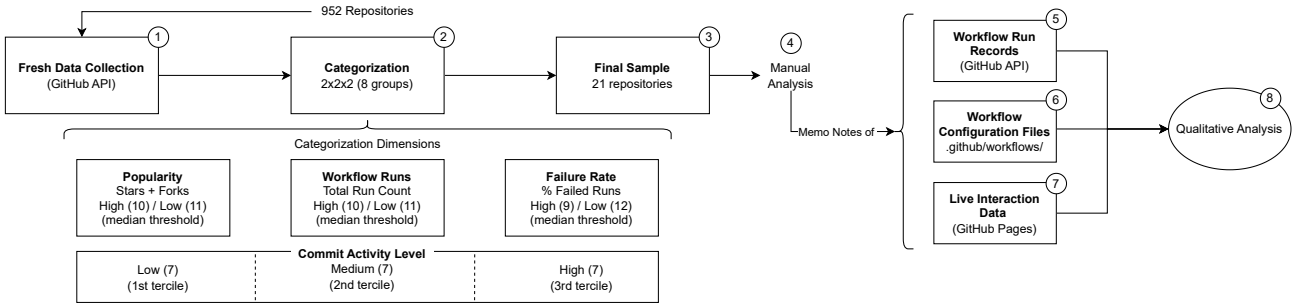

Figure 1: Methodology Overview

repositories using GHA, and (2) ensuring these repositories have accessible execution history. To address these challenges, we begin with an existing dataset while recognizing that GitHub's workflow run retention policy[3] would require us to collect fresh data.

We use the dataset by Bouzenia and Pradel [8], which contains 952 GitHub repositories with workflow run histories. This dataset has a balanced representation of popular repositories (600 with 100+ stars and commits) and less popular ones (352 with fewer than 100 stars). Of the initial 952 repositories, 946 were still publicly available at the time of data collection. We used the GitHub API to collect fresh workflow run data from these 946 repositories, of which 765 contained workflow runs. To collect sufficient data while staying within reasonable API usage limits, we collected up to 1000 run records per repository[4], yielding 258,300 run records collected at the end of 2024 (Figure 1 – ①). We filtered 213,640 non-bot-triggered runs across 742 repositories to analyze for RQ1 (Section 4).

Our data collection pipelines, available in our replication package [26], queried the GitHub API for recent workflow runs, commit histories, and other repository metadata, saving them into a MongoDB database [26] for subsequent analysis.

3.2 Categorization and Sampling Strategy

To capture GHA adoption patterns, we classified the 765 repositories with workflow runs using three attributes: popularity (stars + forks), workflow activity (total run count), and workflow failure rate (percentage of failed runs). We chose these dimensions to represent project scales, CI/CD adoption levels, and failure response contexts essential for our research questions (Figure 1 – ②).

Each attribute was split into high/low categories using median thresholds, creating 8 combinations (2×2×2) as shown in Table 2. Analysis of the population showed relatively balanced distribution across these groups, ranging from 9.4% to 15.7% of repositories per group, outlining that no single usage pattern dominates. While one repository per group could provide insight into each GHA adoption pattern, we selected multiple repositories from most groups based on observed diversity within groups and the need to capture varied adoption approaches within each categorization groups.

Sampling Approach: we selected 21 repositories using purposive sampling to ensure representation across all 8 classification groups (Table 2 and Figure 1 – ③). Rather than proportional sampling, we varied the number of repositories per group (1–5 repositories) based on research value, where some patterns (e.g., low-activity projects with low number of runs and high-activity with high number of runs) needed more examples to understand diverse utilization approaches, and practical coverage to ensure every combination was represented without missing important behavioral patterns.

Figure 2 demonstrates that our sampled repositories span the distribution across all three classification dimensions (popularity, run counts, and run failure rates), ensuring coverage of diverse values. While commit activity was not part of our classification, our sample spans development activity levels capturing projects with varying development intensities.

Sample Characteristics: our final sample ranges from less popular/active projects (72 stars+forks, 7 commits) to more popular/active ones (22,097 stars+forks, 1,466 commits), with 2-1,000 workflow runs and failure rates from 0% to 86.7% (Table 1). This diversity lets us analyze how GHA adoption and developer behaviors vary across different project contexts. We intentionally keep repositories with very few runs, as we are also interested to study project that have workflows configured but rarely use them.

We enriched this data with contextual attributes, including projects' programming languages, commit count, workflow configuration commit count, team and project size indicators to facilitate deeper understanding of project contexts during our sampling and manual analysis phases (Figure 1 – ③). These additional attributes are detailed in Table 3. The complete dataset is available in [26].

3.3 Manual Analysis through Qualitative Coding

To answer RQ2 and RQ3, we conduct a manual analysis. For each of the 21 selected repositories, we analyzed workflow run records (Figure 1 – ⑤), workflow configuration files (Figure 1 – ⑥), and live interaction data[5] (Figure 1 – ⑦), both from our API-collected data (see our dataset [26]) and live data on the repositories' GitHub page. Our manual analysis of each repository includes:

(1) **Workflow file configuration:** analyzing purpose of configuration files (*.github/workflows/*.yml*) and their commit history if and why they were updated.

[3]https://docs.github.com/en/actions/administering-github-actions/usage-limits-billing-and-administration#workflow-run-history-retention-policy

[4]This threshold reflects both API constraints and our pattern-identification goal. Collecting beyond 1000 runs per repository would require more resource and time while providing limited additional insight, as high-volume repositories represent a small minority with predominantly bot-triggered workflows.

[5]Any form of interactions observable from the GitHub website UI, e.g., comments in pull requests, commit messages, active/deactivated workflows.

Table 1: Sampled Repositories Categorization Attributes

ID	Repository	Pop.	Pop.G	Comm.	Comm.G	Runs	Run.G	Run.Fail%	Run.Fail.G
R1	trusttoken/contracts-pre22	450	low	18	low	46	low	0.0	low
R2	nextcloud/ocsms	240	low	32	medium	7	low	0.0	low
R3	ngageoint/geopackage-js	393	low	37	medium	21	low	14.3	low
R4	yandex-cloud/serverless-plugin	72	low	91	medium	37	low	8.1	low
R5	iouAkira/someDockerfile	181	low	15	low	15	low	86.7	high
R6	dotMorten/NmeaParser	351	low	16	low	48	low	47.9	high
R7	nemuTUI/nemu	370	low	20	low	126	low	6.3	low
R8	data-driven-forms/react-forms	399	low	191	high	64	low	4.7	low
R9	komamitsu/fluency	190	low	75	medium	234	high	16.2	high
R10	aws-cloudformation/cfn-lint-vscode	416	low	103	medium	1000	high	2.7	low
R11	boutproject/BOUT-dev	282	low	644	high	1000	high	20.5	high
R12	felix-fly/v2ray-openwrt	823	high	7	low	2	low	0.0	low
R13	Thinkmill/manypkg	974	high	47	medium	142	high	12.7	low
R14	webextension-toolbox/webext-toolbox	800	high	27	low	735	high	21.2	high
R15	prontolabs/pronto	2874	high	14	low	29	low	0.0	low
R16	ai/size-limit	8403	high	160	high	147	high	41.5	high
R17	lc-soft/LCUI	4525	high	209	high	135	low	74.8	high
R18	tsl0922/ttyd	9214	high	73	medium	365	high	0.5	low
R19	RailsEventStore/rails_event_store	1546	high	1151	high	1000	high	0.2	low
R20	BeyondDimension/SteamTools	22097	high	1466	high	1000	high	25.6	high
R21	libcpr/cpr	7599	high	294	high	1000	high	30.4	high

Pop: Popularity (stars + forks), Pop.G: Popularity group, Comm: Total commits since 2023, Comm.G: Total commits group
Run.G: Run group, Run.Fail%: Failure rate percentage, Run.Fail.G: Failure group

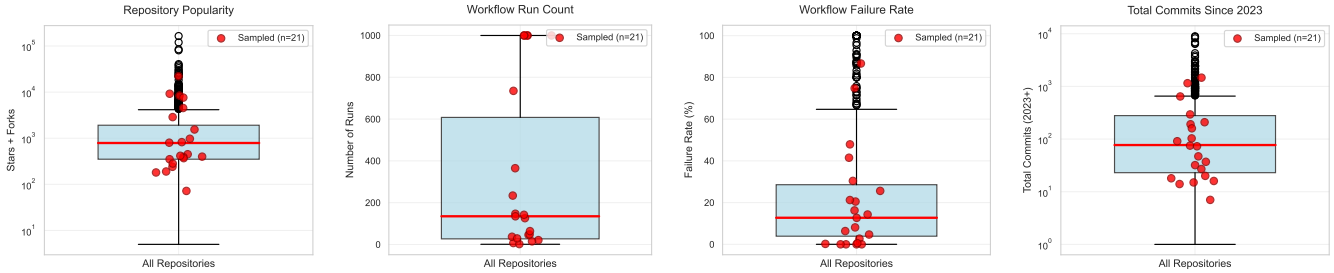

Figure 2: Distribution of sampled repositories (red dots, n=21) across the population. Our sample spans the complete range across (a) repository popularity, (b) workflow run count, (c) failure rates, and (d) commit activity since 2023.

(2) **Workflow run records:** examining workflow run outcomes on main branches and in pull requests, with particular attention to developers' reactions to failures.
(3) **Developer interaction:** analyzing commit messages after failures and discussions in PRs to understand if and how workflow failures triggered actions.
(4) **Repository contextual information:** analyzing characteristics like programming language, team size, maintainer count, development style [22].

Our qualitative analysis (Figure 1 – ⑧) starts with open coding and memo-writing by the first two authors for each repository from the mentioned elements above. Both authors analyzed repositories separately before merging the codes and grouping them into higher-level codes. These were then categorized into broader themes through constant comparison [20] with the original memos. The authors discussed the emerging groups until reaching a negotiated agreement [19]. The analysis required substantial manual effort to scavenge data from multiple sources and interpret the collected information through systematic coding.

Table 2: Repository Sampling: 2×2×2 Classification

Group	Pop	%	Sampled Repositories	n
L-L-L	105	13.7	R1, R2, R3, R4, R7	5
L-L-H	112	14.6	R5, R6	2
L-H-L	86	11.2	R8, R10	2
L-H-H	80	10.5	R9, R11	2
H-L-L	72	9.4	R12, R15	2
H-L-H	94	12.3	R17	1
H-H-L	120	15.7	R13, R18, R19	3
H-H-H	96	12.6	R14, R16, R20, R21	4
Total	**765**	**100**		**21**

Pop=Population count; %=Population percentage
Groups: Popularity-Runs-FailureRate (L=Low, H=High)

3.4 Answering the research questions

Our systematic qualitative coding process of 21 repositories, together with the quantitative analysis of 765 repositories with run records, enabled us to understand workflow adoption beyond their configuration files by examining the rich contextual data surrounding workflow adoption and developer interactions. We use this mixed-methods analysis to answer our three research questions in Sections 4 to 6.

4 RQ1: What are the observed GitHub Actions utilization patterns?

Beginning our analysis, we investigate the broader patterns of how teams utilize GitHub workflows across different repositories. We conduct quantitative analysis across 765 repositories with existing run records, examining the **utilization intensity spectrum** (run

Table 3: Repository Additional Information

ID	Repository	Q.Avg	WF.C	Lang	Age	Trigger Event Distribution	Mo.Gap	U.Runs	B.Runs
R1	trusttoken/contracts-pre22	0.0	0	TypeScript	7.1	sch: 38, pr: 8	8.3	46	0
R2	nextcloud/ocsms	0.0	0	JavaScript	10.3	sch: 7	1.4	7	0
R3	ngageoint/geopackage-js	0.5	4	TypeScript	9.1	push: 19, dyn: 2	2.6	19	2
R4	yandex-cloud/serverless-plugin	0.0	0	TypeScript	4.7	pr: 18, push: 15, dyn: 4	14.2	20	17
R5	iouAkira/someDockerfile	0.7	6	Python	5.4	push: 12, wd: 3	9.3	15	0
R6	dotMorten/NmeaParser	1.2	10	C#	10.9	push: 24, pr: 16, wd: 5, dyn: 3	13.0	48	0
R7	nemuTUI/nemu	0.2	2	C	5.2	push: 72, pr: 54	13.1	126	0
R8	data-driven-forms/react-forms	0.0	0	JavaScript	5.8	pr: 62, dyn: 2	15.1	51	13
R9	komamitsu/fluency	0.5	4	Java	9.3	pr: 196, push: 32, dyn: 6	13.0	50	184
R10	aws-cloudformation/cfn-lint-vscode	2.0	17	JavaScript	6.7	sch: 878, pr: 78, push: 40, rel: 4	9.6	945	55
R11	boutproject/BOUT-dev	8.7	72	C++	11.4	push: 489, pr: 477, dyn: 20, sch: 14	3.3	880	120
R12	felix-fly/v2ray-openwrt	0.5	4	Shell	6.3	push: 2	6.9	2	0
R13	Thinkmill/manypkg	0.5	4	TypeScript	5.4	pr: 82, push: 60	11.7	136	6
R14	webextension-toolbox/webext-toolbox	0.7	6	TypeScript	7.0	pr: 572, sch: 67, push: 49, dyn: 46, wd: 1	15.3	159	576
R15	prontolabs/pronto	0.5	4	Ruby	11.6	pr: 22, push: 7	14.8	29	0
R16	ai/size-limit	1.8	15	JavaScript	7.6	push: 90, pr: 57	12.8	144	3
R17	lc-soft/LCUI	3.2	27	C	12.5	push: 134, pr: 1	13.9	133	2
R18	tsl0922/ttyd	1.0	8	C	8.4	push: 220, pr: 145	13.8	135	230
R19	RailsEventStore/rails_event_store	13.0	108	Ruby	9.8	sch: 960, push: 28, pr: 12	1.9	998	2
R20	BeyondDimension/SteamTools	3.0	25	C#	4.1	push: 791, pr: 98, create: 69, delete: 42	13.1	901	99
R21	libcpr/cpr	4.8	40	C++	9.8	pr: 454, push: 426, dyn: 117, rel: 2, wd: 1	7.1	883	117

Q.Avg: Quarterly avg workflow commits, WF.C: Workflow commits since 2023, Mo.Gap: Months between first and last run record
U/B.Runs: User/Bot runs. Event types: pr: pull_request, sch: schedule, dyn: dynamic, wd: workflow_dispatch, rel: release

counts and failure rates) and **workflow trigger patterns** (events triggering workflow runs). This reveals how frequently run failures occur, whether intensive workflow adoption correlates with failure rates, and which event types and combinations trigger workflow executions in practice.

The workflow trigger analysis shows us what triggers runs based on the run records, rather than their configuration. In this way, we can distinguish whether workflows are activated as part of maintenance automation, or through development-driven triggers like a push or a pull request.

4.1 Utilization Intensity Spectrum

Repositories demonstrate an intensity spectrum from minimal to extensive workflow usage. Pearson correlation [13] analysis reveals a statistically significant negative correlation between run count and failure rate (Correlation coefficient $r = -0.230$, $p < 0.001$ across 765 repositories): as workflow usage increases, failure rates tend to decrease. This high usage can result from two distinct patterns: frequent triggering of workflows due to active development (*activity-intensive*), or comprehensive workflow configurations that execute many jobs per trigger (*configuration-intensive*). In both cases, whether 50 workflow runs result from one event triggering 50 jobs or 50 separate events each triggering one job, the total run volume contributes to the observed statistically significant negative correlation in Figure 3.

To discuss the different trends in Figure 3, we split the resitories into three visually distinct but roughly equally sized buckets: low-usage (<100 runs, 45%), medium-usage (100-499 runs, 26.5%), and high-usage (≥500 runs, 28.5%). These thresholds separate three observable behavioral patterns in Figure 3: the left cluster shows sporadic usage with high failure variability, the right cluster shows sustained automation with more stable failure rates, and the middle represents a transitional pattern[6].

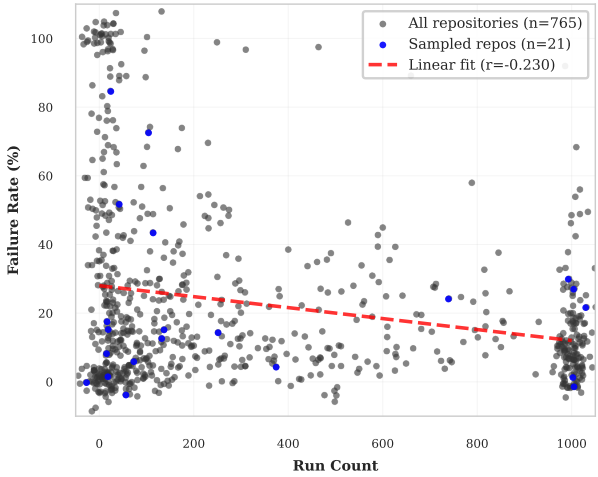

Figure 3: Repositories Run Count vs. Run Failure Rate. Points are jittered horizontally to reduce overplotting.

High-Usage Pattern (28.5% with ≥500 runs): these repositories show high run volumes through two distinct approaches. `RailsEventStore/rails_event_store` exemplifies the *activity-intensive* approach with 1000 runs and 0.2% failure rate during two months, generating volume through frequent development activity supported by 960 scheduled runs[7] and an explicit culture of keeping all runs passing. `boutproject/BOUT-dev` represents the *configuration-intensive* approach with 1000 runs and 20.5% failure rate across 50+ workflow files during 3.3 months, reaching high run record volume through automation breadth while tolerating higher failure rates (e.g., ignoring platform-specific failures in PR #3029, *"LGTM, fedora failure is unrelated"*).

[6]**Note on threshold selection:** The numerical thresholds (100 and 500 runs) presented in this section serve as descriptive categories for interpreting patterns within our specific sample, not as generalizable classification boundaries. These values emerged from visual analysis of our data distribution and are specific to our collection period and sampling approach. The primary contribution is identifying the pattern of systematically decreasing failure rates with increased usage intensity; the specific threshold values are artifacts of our sample and should not be applied to other contexts.

[7]The schedule event allows to trigger a workflow at a scheduled time, like a cron job.

Table 4: Trigger events: distribution and combinations

(a) Top 10 Trigger Event Distribution			(b) Top 10 Trigger Event Comb.		
Event Type	**Runs**	**%**	**Event Combination**	**Repos**	**%**
push	76,822	36	PR + push	238	32
pull_request	76,400	36	push only	85	11
schedule	39,637	19	PR + push + sched	67	9
dynamic	6,387	3	dyn + PR + push	40	5
pull_request_target	4,399	2	PR + push + rel	26	4
issue_comment	2,650	1.2	schedule only	25	3
workflow_run	2,174	1.0	PR only	21	3
issues	1,743	0.8	PR + push + sched + WD	18	2
workflow_dispatch	1,654	0.8	PR + push + WD	15	2
release	616	0.3	dyn + PR + push + sched	15	2

sched = schedule; dyn = dynamic; rel = release; WD = workflow_dispatch

Low-Usage Pattern (45% with <100 runs): these repositories show extreme variability in their run outcomes. Successful minimal usage includes `felix-fly/v2ray-openwrt` (2 runs, 0% failure), `prontolabs/pronto` (29 runs, 0% failure), and `trusttoken/contracts-pre22` (46 runs, 0% failure), exemplifying focused minimal automation. High failure rate minimal usage includes `iouAkira/someDockerfile` (15 runs, 86.7% failure rate): this project creates new workflow files per change and keeps pushing changes till they pass (using it as a tool to publish to docker hub). In `dotMorten/NmeaParser` (48 runs, 31.3% failure rate) the maintainer keeps pushing changes to fix failing workflows after making changes, which results in many workflow failure records. These usage patterns with high failure rates show experimental or trial-and-error approaches.

Medium-Usage Pattern (26.5% with 100-499 runs): these repositories show transitions between low and high-usage patterns. `webextension-toolbox/webextension-toolbox` shows 735 runs with a 21.2% failure rate, where 149 of 156 failures were bot-triggered, the PRs were then closed by the bots (e.g., #1071, #1070, #1068, #1067, #1065, etc.) indicating automation complexity without observable influence on the development. `libcpr/cpr` demonstrates 883 runs with 6.9% failure rate while explicitly accepting failures (PR #1170, *"CI failures are expected and I will fix them soon"*).

The key finding is that average failure rates decrease with higher usage intensity. This pattern suggests that sustained utilization of workflow automation (whether through frequent usage or comprehensive configuration) correlates with lower failure rates, while sporadic or experimental usage results in more variable failure rate.

4.2 Workflow Trigger Patterns

Table 4-a summarizes our analysis of 213 640 non-bot-triggered[8] runs in 742 repositories, showing that teams primarily use development triggers, with selective adoption of scheduled triggers.

Development-Centric Trigger Events: we observe that `push` (36.0% of runs, primary in 332 repositories) and `pull_request` (35.8% of runs, primary in 250 repositories) account for 71.8% of all runs. The most common repository strategy combines both triggers: 238 repositories (32% of all repositories) use the `push` + `pull_request` combination, enabling coverage of both direct commits and PRs (Table 4-b).

Automation Focused Trigger Events: we see that teams selectively adopt automation-focused trigger events. `schedule` triggers account for 18.6% of runs and serve as the primary trigger in 108 repositories, similar to our observations in high-intensity utilizers like `aws-cloudformation/cfn-lint-visual-studio-code` with 878 scheduled runs (out of 1000) for automated schema updates. `dynamic` triggers (3.0% of runs) represent automated services like Dependabot and security scans, serving as primary triggers in 22 repositories but appearing across more repositories as supplementary automation (Table 2-b).

Repository Trigger Diversity Patterns: repositories demonstrate variety in trigger event usage, with an average of 2.5 events used per repository. We also observe that an increased event diversity in a repository correlates with having more run records (Figure 4). However, trigger concentration reveals strategic focus: 284 repositories (38.3%) concentrate more than 80% of their runs in a single trigger event. At the other extreme, repositories like `BeyondDimension/SteamTools` utilize more trigger event diversity (791 `push` + 98 `pull_request` + 69 `create` + 42 `delete`).

The trigger pattern analysis reinforces our intensity findings: higher-usage repositories (≥100 runs) employ diverse trigger combinations, averaging 3.04 unique events, while low-usage repositories (<100 runs) average 1.90 trigger event types. Due to the non-linear relationship visible in Figure 4, we used Spearman's rank correlation [10], which reveals a strong positive correlation between trigger diversity and run count ($\rho = 0.589$, $p < 0.001$). This pattern is consistent: 33.7% of low-usage repositories rely on single triggers compared to 6.3% of higher-usage repositories.

RQ1 Summary: GitHub Actions utilization follows distinct patterns: **High-usage repositories** have low failure rates (2.7–20.5%) with more diverse trigger combinations, while **low-usage repositories** exhibit extreme variability (0–86.7% failure rates) with predominantly single-trigger strategies (33.7% use one trigger event). **Development-centric events triggers** dominate: `push` (36.0%) and `pull_request` (35.8%) account for 71.8% of runs. The negative correlation between run count and failure rates combined with strong correlation between trigger diversity and usage intensity indicates that repositories with sustained workflow usage develop more stable (less failure rate) and comprehensive workflow adoption (diverse trigger events).

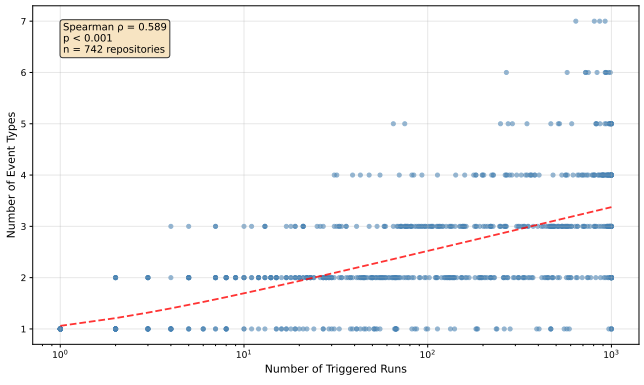

Figure 4: Trigger event diversity increasing as the # of runs increase.

[8]Based on what we see in RQ2, we decided to exclude bot-triggered runs because they showed minimal meaningful reactions compared to user-triggered runs and could skew our results.

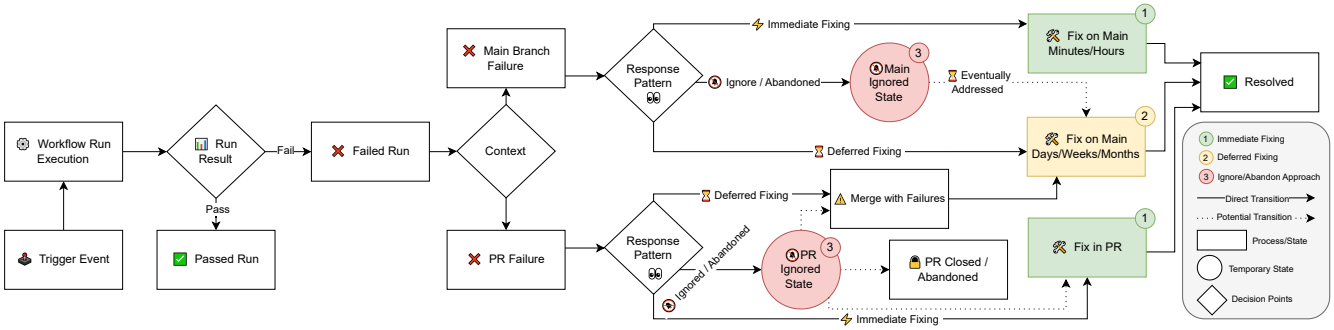

Figure 5: Developer Response Patterns to Workflow Failures in PR and Main Branch Contexts

5 RQ2: How do developers react to GitHub Actions workflow failures?

After understanding the broader workflow utilization of our large sample of 765 repositories, we investigate in more depth how developers react to workflow failures. As this focuses on understanding the context and unique situations of the project, we opt for a qualitative manual analysis.

Our analysis of workflow failure responses across our 21 sampled repositories (Section 3) reveals distinct patterns in how development teams handle GitHub Actions failures. The patterns are not mutually exclusive; projects can exhibit multiple response behaviors. Four repositories in our sampled dataset experienced no failures, so no reactions were observed for them.

We distinguished workflow failures (1) in pull requests (analyzed through PR comments and commits) and (2) on the main branch (tracked through subsequent pushes and timing), as these contexts present different response opportunities and constraints. Figure 5 illustrates how our observed failure reaction patterns occur. We now discuss each pattern.

5.1 Fix-Oriented Response Patterns

5.1.1 ***Immediate Fixing (13 repositories).*** When developers address workflow failures directly within pull requests before merging, or by pushing commits directly to the main branch *immediately* within minutes or hours after (less than 24 hours).

Examples: `webextension-toolbox/webextension-toolbox` shows collaborative fixing: when a contributor's PR fails lint checks, the maintainer (reviewer of the PR) fixes it in the same PR (#866) or guides the contributor to fix it before merging (PR #890). In `trusttoken/contracts-pre22` the contributor makes multiple commits to resolve failed run issues before merging (PR #1264). In `data-driven-forms/react-forms` the contributor gets feedback from the PR reviewer and keeps pushing fix changes until all check runs pass before a merge (PR #1492). Developers also show immediate fixing behavior when directly pushing to the main branch of a repository. For example, in `ai/size-limit` when there is a commit with failed runs on the main branch, we see cases of immediate fixing in the subsequent commit(s) happening *shortly* after, e.g., commit 058b1f1 workflow runs pass 3 minutes after a commit with a failed run. In `BeyondDimension/SteamTools` we observe instances of fixing on the same day (commit 078d462, and commit 3633915).

`iouAkira/someDockerfile` shows the same behavior but with a different goal: the user keeps pushing changes until the run for Docker image builds passes, effectively treating GitHub Actions as a deployment service to push images to the Docker Hub. `RailsEventStore/rails_event_store` demonstrates an explicit culture of keeping all runs passing where failed commits are followed by immediate fixing commits (we did not observe a deferred fix approach for this repository). `aws-cloudformation/cfn-lint-visual-studio-code` shows a culture of ensuring workflows run successfully. The few run failures that we observed were fixed during the related PR.

Summary: immediate fixing, the most prevalent pattern observed in 76% of repositories with run failures, represents a proactive approach where teams resolve workflow failures within minutes to hours. This pattern manifests both in the pull-based development model [22, 23, 39] and in projects that directly commit to the main branch (Figure 5 – ①).

5.1.2 ***Deferred Fixing (6 repositories).*** Occurs when teams accept temporary failures in their main branch and address them in subsequent commits. We distinguish between immediate and deferred fixes based on timing. If the failure occurs on the main branch and the commit that resolves the workflow failure happens days (≥ 24 hours) later, rather than within minutes or hours, we consider it a deferred fix.

Examples: in `libcpr/cpr` (PR #1170) a maintainer states *"The CI failures are expected and I will fix them soon on master,"* showing a deferred fixing approach to address failures post-merge rather than blocking development velocity (later fixed in PR #1173). `komamitsu/fluency` exhibits deferred fixing where a Dependabot version bump caused CI failures (commit c870a74), but the maintainer addressed the issue almost 2 months later through a separate PR (#915).

In `BeyondDimension/SteamTools` we observed multiple deferred fixing examples: PR #3469 merged with failed runs and was fixed 2 days later (commit 7b9ba5e), and a prolonged CI failure period of two months (between commits 150f93e and c087a02) caused by an UI framework upgrade, was eventually resolved through workflow configuration updates.

The deferred fixing approach spans varying timeframes, from days to months, reflecting different tolerances for temporary failures. `yandex-cloud/serverless-plugin` exhibits extended delays with NodeJS version-related failures taking almost a month after the first failure to be addressed through dependency upgrades, showing tolerance for dependency-related issues. `libcpr/cpr` demonstrates

a deferred fixing with a 4-month gap between failed and successful workflow runs on master (between commit 372b54c and commit a9466f7), indicating acceptance of non-critical deployment failures until the next release cycle.

Summary: the deferred fixing approach reflects tolerance for temporary failures, with resolution timeframes spanning days to months. Teams employing this pattern appear to distinguish critical path failures requiring immediate attention and issues that can be addressed during planned maintenance. This prioritizes development velocity over passing workflow runs, accepting temporary failed runs in exchange for unblocked progress (Figure 5 – ②).

5.1.3 ***Ignore/Abandon Approach (4 repositories).*** Describes workflow failures that receive no addressing action, representing a temporary state where teams either deliberately accept certain failures, face resource constraints preventing attention, or ultimately remove (or disable) failed workflows. In PRs, failures are ignored if they receive no reaction comments or fixing commits, or if the PR is closed/abandoned without resolution. On the main branch, failures remain ignored if they persist through subsequent commits without addressing action (Figure 5 – ③). Our study window is a snapshot: ignored failures may later transition to the deferred fixing pattern.

Examples: several repositories demonstrate explicit decisions to ignore certain workflow failures with clear justifications. `boutproject/BOUT-dev` demonstrates platform-specific ignoring where the reviewer explicitly states *"LGTM, fedora failure is unrelated"* when merging PRs with Fedora platform CI failures (#3029), and also states *"I am happy to remove the coverage run. I have not looked at the results in years..."* showing ignoring test coverage failures (#3067). This indicates an approach of dismissing environment-specific issues deemed non-critical. Similarly, in `data-driven-forms/react-forms` PR #1449 was merged while CodeCov coverage check failed, indicating tolerance for specific quality assurance metric failures. Another example is `ngageoint/geopackage-js`, where sometimes the last commit remains failing for a while (e.g., commit 3981bf0), potentially representing a conscious trade-off between CI/CD hygiene and development velocity.

Complete abandonment of workflow failures occurs when teams determine to remove or disable the workflow. `BeyondDimension/SteamTools` provides a clear example with marking `dotnet.yml` and `xamarin.yml` as *disabled* after persistent failures. The repository also shows workflows abandonment: the *publish* workflow never ran successfully after commit b40dac0, with the team stopping attempts to fix (till the time of our study). `trusttoken/contracts-pre22` demonstrates migration from GHA to CircleCI (abandoning GHA) and then again abandoning CircleCI by removing the workflow; also disabling their Coveralls test workflow. `iouAkira/someDockerfile` has 13 different workflow configurations but only 2 with run records, suggesting that most workflows were effectively disabled after initial setup.

Summary: the ignore/abandon approach reflects three distinct scenarios: deliberate acceptance of failed runs, resource constraints that prevent immediate attention to failed runs, or removal of workflows causing failures. Teams may ultimately resolve ignored failures through three pathways: transitioning to immediate or deferred fixing, continuing to leave them unaddressed, or removing failing workflows entirely. The temporary nature of this classification highlights that ignored failures represent a point-in-time observation rather than a permanent team strategy, with resolution approaches potentially evolving as project priorities or resources change.

Table 5: Workflow utilization metrics (from RQ1).

Characteristic	Category	N	Runs	Fail %	Div.
Activity	High	7	620.9	28.2	3.1
	Medium	7	258.0	7.8	2.4
	Low	7	143.0	23.2	2.6
Workflow Ev.	High	5	827.0	30.3	3.6
	Moderate	5	459.0	22.8	3.4
	Minimal	7	81.3	19.5	2.0
	No	4	38.5	3.2	2.0
Team Size	Solo	3	339.0	29.8	2.3
	1 maint.	10	282.6	23.9	2.8
	Multi maint.	8	413.8	10.8	2.8
Dev. Style	Direct push	2	8.5	43.3	1.5
	Mixed	11	343.9	22.7	2.7
	Pull-based	8	419.1	9.7	3.0
Popularity	High	10	455.5	20.7	2.8
	Low	11	236.2	18.9	2.6

Runs = average workflow runs; Fail % = average failure rate; Div. = average trigger event diversity

RQ2 Summary: our analysis of 21 repositories reveals three distinct response patterns: immediate fixing (13 of repositories) where teams resolve failures within minutes to hours, reflecting a commitment to passing continuous integration; deferred fixing (6 repositories) where teams strategically tolerate temporary failures, distinguishing between critical issues and those addressable in the future; and ignore/abandon (4 repositories) where failures receive no immediate action due to deliberate acceptance, potential resource constraints, or eventual workflow removal. These non-mutually exclusive patterns demonstrate that teams make nuanced prioritization decisions potentially based on failure context, project constraints, and strategic considerations.

6 RQ3: What relationships exist between project characteristics and GitHub Actions utilization patterns?

Understanding how project characteristics influence GitHub Actions utilization is fundamental for developing context-aware CI/CD research and practice [27]. In this section, we combine the insights from our quantitative analysis (Section 4) and in-depth qualitative analysis (Section 5). We examine relationships between project characteristics (popularity, development style, team size, workflow evolution, and activity level) relate to utilization patterns (RQ1) and failure reactions (RQ2). Given our limited sample size from our qualitative analysis, we deliberately emphasize these as hypotheses that represent a research agenda for future quantitative validation in large-scale mining studies.

6.1 Project Characteristics and Workflow Utilization Patterns

Table 5 presents workflow utilization metrics grouped by project characteristics. These observed patterns inform our hypotheses (H1-H3) for future validation.

Activity Level: high-activity projects show the most intensive workflow usage with 620.9 average runs and the highest event diversity (3.1), but also experience higher failure rates (28.2%). Medium-activity projects achieve the lowest failure rates (7.8%) despite moderate usage (258 runs). Low-activity projects show limited workflow adoption (143 runs) with surprisingly higher failure rates (23.2%), potentially due to experimental or one-off task usage (e.g. `iouAkira/someDockerfile`).

Workflow Evolution: projects with high workflow evolution demonstrate the most intensive usage patterns (827 runs) and highest event diversity (3.6), but also highest failure rates (30.3%). Projects with no workflow evolution show minimal usage (38.5 runs) with the lowest failure rates (3.2%). Moderate evolution balances usage (459 runs) with failure rates (22.8%) closer to the overall average (19.7%).

Team Size: multi-maintainer projects have the highest run counts (413.8), yet lowest failure rates (10.8%). Solo projects demonstrate moderate usage (339 runs) but high failure rates (29.8%), while single-maintainer projects show the lowest usage (282.6 runs) with intermediate failure rates (23.9%). Event diversity is consistent across team size (2.3-2.8), suggesting trigger complexity is independent of team structure.

Development Style: pull-based development has the highest workflow usage (419.1 runs) and event diversity (3) with the lowest failure rates (9.7%). Direct push approaches show minimal workflow adoption (8.5 runs) with high failure rates (43.3%). Mixed approaches balance moderate usage (343.9 runs) with moderate failure rates (22.7%).

Popularity: high-popularity projects show nearly double the workflow usage of low-popularity projects (455.5 vs. 236.2 runs) with comparable failure rates (20.7% vs. 18.9%), suggesting that project popularity may drive workflow adoption intensity. Event diversity shows minimal variation (2.8 vs. 2.6), indicating that trigger patterns are consistent regardless of popularity.

Hypothesis H1: pull-based development is linked to lower workflow failure rates compared to direct push approaches.

Hypothesis H2: projects with higher workflow evolution intensity experience higher failure rates, suggesting that frequent configuration changes are linked to increased workflow failures.

Hypothesis H3: team size is negatively linked to workflow failure rates, with projects having multiple maintainers showing lower failure rates than solo projects, and single-maintainer projects falling between these extremes.

6.2 Project Characteristics and Workflow Failure Reaction Patterns

Table 6 summarizes how projects with different characteristics react to workflow failures. These observed patterns inform our hypotheses (H4 and H5) for future validation.

Activity Level: projects with high activity levels demonstrate the most diverse failure reaction strategies, with 71.4% employing immediate fixes, and 57.1% using both deferred fixes and ignore approaches. This suggests that actively developed projects have more flexibility in their failure response strategies. In contrast, projects with low activity exclusively use immediate fixes when they address failures, never showing deferred or ignore fixing among our studied projects, potentially reflecting their more focused scope and simpler workflows.

Table 6: Workflow failure reaction patterns (from RQ2)

Characteristic Categories	Total	Immediate Fixing (%)	Deferred Fixing (%)	Ignore /Abandon (%)
Activity Level				
High	7	5 (71.4)	4 (57.1)	4 (57.1)
Medium	7	4 (57.1)	2 (28.6)	0 (0.0)
Low	7	4 (57.1)	0 (0.0)	0 (0.0)
Workflow Evolution				
High	5	3 (60.0)	4 (80.0)	4 (80.0)
Moderate	5	4 (80.0)	0 (0.0)	0 (0.0)
Minimal	7	4 (57.1)	1 (14.3)	0 (0.0)
No	4	2 (50.0)	1 (25.0)	0 (0.0)
Team Size				
Solo project	3	2 (66.7)	0 (0.0)	0 (0.0)
1 maintainer	10	5 (50.0)	3 (30.0)	2 (20.0)
Multiple maintainers	8	6 (75.0)	3 (37.5)	2 (25.0)
Development Style				
Direct push	2	1 (50.0)	0 (0.0)	0 (0.0)
Mixed	11	8 (72.7)	2 (18.2)	2 (18.2)
Pull-based	8	4 (50.0)	4 (50.0)	2 (25.0)
Popularity				
High	10	6 (60.0)	3 (30.0)	3 (30.0)
Low	11	7 (63.6)	3 (27.3)	1 (9.1)

Note: Four repositories had no failures. Workflow failure reaction patterns are not mutually exclusive, so percentages do not add up to 100%.

Workflow Evolution: a pattern emerges where projects with frequent workflow configuration updates show the highest rates of both deferred fixing (80%) and ignoring failures (80%), while projects with moderate evolution demonstrate the opposite behavior: 80% immediate fixing with no deferred or ignore strategies. This counterintuitive finding suggests that frequent workflow changes may create technical debt that teams manage through selective attention to failures.

Team Size: projects with multiple maintainers show a higher rate of immediate fixes (75%), compared to single-maintainer projects (50%). Solo projects never employ deferred or ignore strategies. This suggests that projects with multiple maintainers are more likely to have diverse failure response strategies, potentially reflecting prioritization, distributed responsibilities, and project complexity.

Development Style: development approaches combining PRs with direct pushes (mixed) are more likely to do immediate fixes (72.7%), while pull-based development shows equal distribution between immediate (50.0%) and deferred fixes (50.0%).

Popularity: repository popularity shows little to no relation with failure reaction patterns, with both high and low popularity projects demonstrating similar distributions across approaches. This indicates that reactions to workflow failures are independent of repository popularity.

Hypothesis H4: projects with high workflow evolution rates are more likely to employ selective failure fixing strategies (deferred and ignore approaches), potentially indicating that rapid configuration changes create technical debt that teams should prioritize selectively.

Hypothesis H5: projects with multiple maintainers are more likely to employ immediate fixing strategies than single-maintainer projects.

7 Discussion

7.1 Collaborative vs. Direct Fixing in Workflow Failures

Our observations reveal that while contributors may resolve workflow failures independently, maintainer involvement creates a duality in fixing approaches: maintainers either (1) guide external contributors to fix issues themselves, or (2) directly resolve failures within the PR. This pattern can reflect decisions about knowledge transfer when maintainers choose to intervene in case of workflow run failure in a pull request. When maintainers provide guidance, failures become learning opportunities for contributors to understand project context [33]. When maintainers fix directly, development velocity is preserved but the knowledge sharing value is lost, potentially affecting contributor retention.

7.2 Local Optimization vs. Collective Costs

The deferred fixing approach that we observe in nearly 1/3 of our sampled repositories, reveals an asymmetry: while teams may make locally rational decisions to defer fixes, this can impose hidden costs on the development community.

We saw teams may make calculated decisions about failure priority, treating certain workflow issues as manageable technical debt, e.g., deferred fixing. Maintainers may reasonably defer fixes for infrastructure issues or dependency problems that do not block core development workflows. However, this local optimization can create problematic downstream effects. When subsequent contributors encounter failing builds, they may assume their changes caused the failure, leading to wasted debugging effort. This *failure contamination* represents an asymmetric cost distribution where one team's deferred fix may generate time waste elsewhere.

Additionally, normalization of failing workflows, may diminish contributor confidence in GHA workflows and reduce sensitivity to genuine new issues.

7.3 The Configuration-Usage Gap

Our approach of examining workflow run data rather than static configuration files, reveals methodological considerations for GHA workflows research. While most previous studies analyzing GHA adoption focused on workflow configurations (Section 2), our study uncovered cases where configurations did not reflect actual adoption: disabled workflows (`nextcloud/ocsms`, with CodeQL scans disabled and no runs in years), unused configurations (`iouAkira/someDockerfile` with 13 configurations but only 2 active), and abandoned workflows. This suggests that configuration-based analyses may overestimate actual workflow adoption and may miss the nuanced reality.

7.4 Ideas for Enhancing GitHub Actions

Our findings illuminate areas where the current GHA platform can better support distributed development coordination:

Intelligent Failure Context: providing automated failure attribution and historical context to help contributors distinguish between inherited issues and newly-introduced problems.

Adaptive Mentoring Support: enhanced tooling could automatically generate project-specific guidance for common workflow failures, reducing maintainer mentoring burden while preserving contributor learning opportunities [2].

Usage-Aware Analytics: moving beyond configuration to usage-based insights would help teams understand automation adoption patterns and identify underutilized workflows.

7.5 Threats to Validity

Internal validity: Subjective interpretation of developer behavior was mitigated through dual coding and negotiated agreement between two authors. **External validity:** Our 21 repositories, while diverse across key dimensions, may not represent all GHA usage patterns. We frame RQ3 findings as hypotheses requiring future large-scale validation. **Construct validity:** Our observation of reactions may not fully reflect underlying team intentions. The 1000-run collection limit may shorten data for highly active projects. The thresholds used in RQ1 are subjective and for interpretation only.

8 Conclusion and Future Work

We examined GitHub Actions adoption through workflow execution data and developer behavior, analyzing 21 repositories to understand how teams utilize automation in practice. Analyzing 765 repositories (RQ1) revealed that higher usage intensity correlates with lower failure rates: repositories with more runs have lower failure rates and use diverse triggers, while repositories with fewer runs show high variability with predominantly single-trigger strategies. Building on these findings, we identified three distinct failure response patterns (RQ2): *immediate fixing*, resolving failures within hours, *deferred fixing*, tolerating temporary failures for days to months, and *ignore/abandon* approaches, where failures receive no immediate action or lead to workflow removal. These patterns indicate teams make strategic prioritization decisions based on context. Finally, our mixed-methods analysis, combining quantitative metrics from 21 repositories with qualitative examination of their contexts, yielded five hypotheses for future validation (RQ3): workflows with more changes show more selective fixing strategies; multiple maintainers enable higher immediate fixing rates; pull-based development shows lower workflow failure rates; workflows with more changes have higher failure rates; and larger teams experience fewer workflow failures.

In terms of future work, we propose to (1) perform a *large-scale validation* across diverse repositories. Statistical analysis should establish the generalizability of relationships between project characteristics and workflow utilization patterns identified in our study. Additionally, (2) we need to better understand the cost-benefit relationship of different workflow strategies, e.g., the costs of a deferred fixing pattern. Finally, (3) we intend to investigate AI-assisted workflow guidance systems, and intelligent failure attribution to reduce the cost of fixing workflow failures.

Data Availability

All the data, data collection pipelines, and analyses are available in our replication package [26].

Acknowledgments

This research was partially funded by the Dutch science foundation NWO through the Vici "TestShift" grant (No. VI.C.182.032).

References

[1] Jessy Ayala and Joshua Garcia. 2023. An empirical study on workflows and security policies in popular github repositories. In *2023 IEEE/ACM 1st International Workshop on Software Vulnerability (SVM)*. IEEE, 6–9.

[2] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, 712–721.

[3] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 470–481.

[4] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: an explorative analysis of travis CI with github. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 356–367.

[5] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Travistorrent: synthesizing travis CI and github for full-stack research on continuous integration. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 447–450.

[6] Giacomo Benedetti, Luca Verderame, and Alessio Merlo. 2022. Automatic security assessment of GitHub Actions workflows. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, 37–45.

[7] Al Bessey et al. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53, 2, 66–75.

[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 (ICSE)*. ACM, 25:1–25:12.

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

[10] Jacob Cohen, Patricia Cohen, Stephen G West, and Leona S Aiken. 2013. *Applied multiple regression/correlation analysis for the behavioral sciences*. Routledge.

[11] 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, 111827.

[12] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. 2022. On the use of GitHub Actions in software development repositories. In *IEEE International Conference on Software Maintenance and Evolution, (ICSME)*. IEEE, 235–245.

[13] Frederik Michel Dekking, Cornelis Kraaikamp, Hendrik Paul Lopuhaä, and Ludolf Erwin Meester. 2005. *A Modern Introduction to Probability and Statistics: Understanding why and how*. Vol. 488. Springer.

[14] Hassan Onsori Delicheh, Alexandre Decan, and Tom Mens. 2023. A preliminary study of GitHub Actions dependencies. In *SATToSE*, 66–77.

[15] Hassan Onsori Delicheh and Tom Mens. 2024. Mitigating security issues in GitHub Actions. In *Proceedings of the 2024 ACM/IEEE 4th International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) and 2024 IEEE/ACM Second International Workshop on Software Vulnerability*, 6–11.

[16] Omar Elazhary, Margaret-Anne D. Storey, Neil A. Ernst, and Andy Zaidman. 2019. Do as I do, not as I say: do contribution guidelines match the GitHub contribution process? In *2019 IEEE International Conference on Software Maintenance and Evolution, (ICSME)*. IEEE, 286–290.

[17] Omar Elazhary, Colin Werner, Ze Shi Li, Derek Lowlind, Neil A. Ernst, and Margaret-Anne Storey. 2022. Uncovering the benefits and challenges of continuous integration practices. *IEEE Transactions on Software Engineering*, 48, 7, 2570–2583. DOI: 10.1109/TSE.2021.3064953.

[18] M. Fowler and M. Foemmel. [n. d.] Continuous integration. [Online; accessed 29-May-2025]. (). https://tinyurl.com/ycbl2uhj.

[19] D. Randy Garrison, Martha Cleveland-Innes, Marguerite Koole, and James Kappelman. 2006. Revisiting methodological issues in transcript analysis: negotiated coding and reliability. *Internet High. Educ.*, 9, 1, 1–8.

[20] Barney Glaser and Anselm Strauss. 2017. *Discovery of grounded theory: Strategies for qualitative research*. Routledge.

[21] 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.

[22] Georgios Gousios and Andy Zaidman. 2014. A dataset for pull-based development research. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (MSR 2014). ACM, 368–371.

[23] Georgios Gousios, Andy Zaidman, Margaret-Anne D. Storey, and Arie van Deursen. 2015. Work practices and challenges in pull-based development: the integrator's perspective. In *37th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 358–368.

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

[25] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs? In *International Conference on Software Engineering (ICSE)*. IEEE, 672–681.

[26] Ali Khatami, Carolin Brandt, and Andy Zaidman. 2026. Replication package for "Beyond the YAML File: Understanding Real-World Github Actions Workflow Adoption. (2026). DOI: 10.5281/zenodo.18258226.

[27] Ali Khatami, Carolin Brandt, and Andy Zaidman. 2024. Software quality assurance analytics: enabling software engineers to reflect on QA practices. In *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, 189–200.

[28] Ali Khatami, Cédric Willekens, and Andy Zaidman. 2024. Catching smells in the act: A github actions workflow investigation. In *International Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 47–58.

[29] Ali Khatami and Andy Zaidman. 2024. State-of-the-practice in quality assurance in Java-based open source software development. *Software: Practice and Experience*, 54, 8, 1408–1446.

[30] 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.

[31] Eriks Klotins, Tony Gorschek, Katarina Sundelin, and Erik Falk. 2022. Towards cost-benefit evaluation for continuous software engineering activities. *Empirical Software Engineering*, 157, 6.

[32] 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.

[33] Zhixing Li, Yue Yu, Tao Wang, Shanshan Li, and Huaimin Wang. 2022. Opportunities and challenges in repeated revisions to pull-requests: an empirical study. *Proc. ACM Hum.-Comput. Interact.*, 6, CSCW2, Article 317, (Nov. 2022), 35 pages.

[34] Pooya Rostami Mazrae, Alexandre Decan, and Tom Mens. 2024. Gawd: a differencing tool for github actions workflows. In *Proceedings of the 21st International Conference on Mining Software Repositories*, 682–686.

[35] 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, 52.

[36] Jadson Santos, Daniel Alencar da Costa, Shane McIntosh, and Uirá Kulesza. 2025. On the need to monitor continuous integration practices. *Empirical Software Engineering*, 30, 5, (June 2025), 47 pages.

[37] 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*. ACM, 121–130.

[38] Pablo Valenzuela-Toledo and Alexandre Bergel. 2022. Evolution of github action workflows. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 123–127.

[39] Erik van der Veen, Georgios Gousios, and Andy Zaidman. 2015. Automatically prioritizing pull requests. In *12th IEEE/ACM Working Conference on Mining Software Repositories (MSR)*. IEEE, 357–361.

[40] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 805–816. ISBN: 9781450336758.

[41] Mairieli Wessel, Joseph Vargovich, Marco Aurélio Gerosa, and Christoph Treude. 2023. Github actions: the impact on the pull request process. *Empir. Softw. Eng.*, 28, 6, 131.

[42] Yang Zhang, Yiwen Wu, Tingting Chen, Tao Wang, Hui Liu, and Huaimin Wang. 2024. How do developers talk about GitHub Actions? Evidence from online software development community. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. ACM.