On the GitHub Actions Language: Usage, Evolution, and Workflow Reliability

Aref Talebzadeh Bardsiri
Software Engineering Lab
University of Mons
Mons, Belgium
aref.talebzadehbardsiri@umons.ac.be

Alexandre Decan
F.R.S.-FNRS Research Associate
Software Engineering Lab
University of Mons
Mons, Belgium
alexandre.decan@umons.ac.be

Tom Mens
Software Engineering Lab
University of Mons
Mons, Belgium
tom.mens@umons.ac.be

***Abstract*—Developers often struggle with maintaining GitHub Actions workflow configurations in GitHub-hosted repositories, with recent studies showing frequent execution failures. This paper empirically explores how the adoption and evolution of GitHub Actions language constructs impacts workflow reliability and maintainability. To do so, we quantitatively analyse 260K workflows from 49K GitHub repositories to understand how they are used in practice and how their usage has evolved from July 2019 to August 2025. We identify 197 language constructs available in the GitHub Actions language and map them to 14 features reflecting workflow capabilities. We observe that only a small set of constructs and features are used very frequently, and that larger and more complex workflows are associated with higher failure rates and more maintenance effort. We identify specific features that are more likely to be linked with reliability and maintainability risks. These insights can help practitioners and researchers improve their understanding and usage of the GitHub Actions language, which can help in improving and sustaining workflow automation practices.**

***Index Terms*—GitHub Actions, CI/CD, workflow automation, workflow reliability**

I. Introduction

With the increasing demand for efficient and high-quality software systems, CI/CD practices have become mainstream in software projects to streamline their development and deployment pipelines. CI/CD services automate repetitive tasks such as building code, running tests, performing quality and security checks, and deploying applications. They are an integral part of collaborative software development because they enhance productivity, improve efficiency, and reduce the likelihood of human errors [1].

In the past, different CI/CD services (e.g., Travis, CircleCI and Jenkins) were frequently used in GitHub repositories. Since the public release in 2019 of GitHub's integrated CI/CD solution called GitHub Actions (hereafter shortened to GHA), it has become the most popular CI/CD tool on GitHub [2]. GHA allows repository maintainers to automate workflows through YAML-based configuration files. For writing these workflows, GHA provides a rich set of language constructs (i.e., keys, structures, values, etc.).[1] It can be considered as a domain-specific language [3] that allows workflow maintainers to define workflows, jobs, steps, and more.

Its seamless integration with GitHub, its large marketplace of Actions, and its generous free plan for running workflows for public repositories, have made GHA a compelling choice for many developers [2], [4]. However, developers have highlighted that using GHA workflows comes with multiple challenges [5], [6], such as the difficulty in writing, testing, and debugging workflow files. Ghaleb et al. [7] reported that GHA workflows are among the most complex CI/CD automation services due to their high maintenance effort, while Zheng et al. [8] observed that GHA workflow files frequently fail during execution, highlighting challenges related to reliability and efficiency. These observations suggest that GHA syntax and semantics may be poorly understood or mastered by workflow maintainers, or that tool support is below par.

Despite the widespread adoption of GHA, few studies have empirically analyzed the language constructs used in GHA workflow files, their usage patterns, and their evolution over time. While the aforementioned challenges suggest that complex CI/CD configurations have a negative impact on software quality, there is a lack of quantitative evidence at scale linking the specific structural growth of GHA workflow files to their reliability and maintainability. Addressing this gap is crucial to help practitioners manage configuration complexity and establish concrete best practices for CI/CD automation.

This paper therefore presents a large-scale quantitative study of the GHA language, its usage, and its direct impact on workflow maintainability and reliability. We leverage a dataset of 260K workflows from 49K GitHub repositories, tracking their evolution over a six-year time span from July 2019 to August 2025. We conduct statistical analyses on a subset of these workflows to evaluate how their structural size and feature usage correlate with reliability and maintainability issues. We aim to answer the following research questions:

RQ_0 *What are the constructs and features of the GHA language?* A first step towards understanding the usage of GHA is to identify the constructs provided by its language. We show that GHA provides a rich language of 197 distinct constructs, which we group into 14 high-level features.

[1] https://docs.github.com/en/actions/reference/workflows-and-actions/workflow-syntax

RQ_1 *Which constructs are used in practice?* Understanding the usage frequency of the language constructs can reveal which of them are central to workflow configurations and which ones tend to be more specialized. We find that only a subset of them are frequently used, that most workflows tend to use a wide variety of constructs, and that several constructs are being used repeatedly within a same workflow.

RQ_2 *Which features are used in practice?* Similarly, we identify which features are commonly used and which ones are rarely observed in practice. In addition, we analyze to what extent workflows make use of each feature in terms of its available constructs, and which constructs are most frequently employed in workflow configurations.

RQ_3 *How does the GHA language usage evolve over time?* We analyze how the usage of GHA constructs and features has changed during the six-year observation period. We find that the number of paths in workflows tends to increase over time by a factor of two to three. In contrast, the number of constructs used in workflows only increases slightly, and the number of features used in workflows tends to remain stable. This suggests that increases in workflow size are not due to the usage of more features, but rather to the reuse of the same set of constructs in more places.

RQ_4 *To what extent does workflow size impact reliability and maintainability?* We analyze the relationship between workflow size and workflow reliability and maintainability. We show that larger workflows are significantly more prone to execution failures, require more maintenance effort, have lower overall availability, and take longer to repair after a failure. In addition, we show that the use of specific features is more strongly associated with reliability and maintainability issues than other features.

The remainder of this article is structured as follows. Section II introduces the core concepts and terminology used in the paper. Section III explains the data extraction. Sections IV to VIII address the research questions. Section IX presents the related work. Section X discusses the threats to validity, and Section XI concludes the paper and discusses the findings.

II. Background

Like any CI/CD service, GHA enables GitHub repository maintainers to configure workflows to automate the building, testing, analysis and deployment of their software projects. Developers can also use GHA to automate many other activities, such as managing issues and pull requests, sending notifications, and more.[2] Its tight integration with GitHub makes GHA a popular choice among developers [4].

GHA workflow configurations use the YAML file format and are stored in the *.github/workflows* directory of a GitHub repository. Figure 1 provides an example of a workflow configuration to automate the building and testing of a Node.js project. Each workflow has a name, which can either be defined in the file (line 1) or inferred from the filename, if not explicitly defined. Workflows can be triggered by a wide range of *events* (e.g., push, pull request, schedule) specified by the on key (line 2). Lines 3-4 declare that the workflow is triggered by a push event on the main branch. The workflow defines two *jobs* labeled build (lines 8-15) and test (lines 16-20). Jobs can execute in parallel on one or more *runners* (runs-on, lines 12 and 18) in GitHub-hosted or self-hosted virtual environments. Each job declares one or more steps (lines 13-15 and lines 19-20). A run step (lines 15 and 20) allows maintainers to execute a sequence of shell commands. A uses step (line 14) executes a reusable component, called *Action*, sourced from public GitHub repositories and often found on the GitHub Marketplace.[3] Steps can even reuse entire workflows.[4] One can define different configurations for a job through the *matrix strategy* (strategy and matrix on lines 9-11), allowing the same job to be run the same job in different environments, operating systems or language versions (e.g., the user-defined os variable defined on line 11 and used on line 12). Maintainers can also specify the *permissions* granted to a workflow (line 5), controlling its access to the repository content (line 6), to the issue tracker, to pull requests, etc.

```
1  name: CI
2  on:
3    push:
4      branches: [main]
5  permissions:
6    contents: read
7  jobs:
8    build:
9      strategy:
10       matrix:
11         os: [ubuntu-latest, windows-latest]
12     runs-on: ${{ matrix.os }}
13     steps:
14       - uses: actions/checkout@v3
15       - run: npm ci
16   test:
17     needs: build
18     runs-on: ubuntu-latest
19     steps:
20       - run: npm test
```

Fig. 1. Example of a GHA workflow configuration.

The YAML format of workflow configurations imposes a hierarchical structure of key-value pairs, lists, and nested elements. A *path* in a workflow refers to a specific element within this structure, starting from a top-level key. We separate the different levels of the hierarchy by dots. For example, the path *jobs*.*build*.*steps [0]*.*uses* refers to the Action used by the first step (at index 0) of the *build* job (line 14 in Figure 1).

The GHA syntax allows for user-defined keys in various places. Figure 1 shows two examples of user-defined keys for jobs (build on line 8 and test on line 16) and for matrix variables (os on line 11). User-defined keys can also be used for environment variables, Action parameters, etc.[5]

[2]https://docs.github.com/en/actions/get-started/understand-github-actions

[3]https://github.com/marketplace?type=actions

[4]https://docs.github.com/en/actions/sharing-automations/reusing-workflows

[5]We refer to the online documentation of GHA for more details.

To facilitate analysing and comparing paths across different workflows using different user-defined keys, we introduce the concept of *constructs*. A *construct* represents an abstract workflow path where user-defined keys and list indices are replaced by generic placeholders. For example, the concrete path *jobs*. *build*. *steps [0]*. *uses* is abstracted to the construct *jobs*.<*id*>.*steps [*]*. *uses*, where <*id*> is a placeholder for the job identifier and *[*]* abstracts the actual index of the step.

III. Data Extraction

In order to study GHA language usage evolution, we need a large collection of workflow histories belonging to a diverse set of software development repositories on GitHub, excluding experimental or personal repositories. Cardoen et al. [9] provide such a dataset, containing the history and content of 267K workflows from 49K software development repositories, covering the period from July 2019 to August 2025, and containing over three million workflow snapshots. The repositories have been selected based on their popularity (at least 100 stars), activity (at least 300 commits) and recency (at least one commit after August 25th, 2024). We relied on version 2025-10-09 of this dataset to conduct our study.

Since our focus is on the actual usage of the GHA language, we excluded 152,380 *invalid* workflow snapshots (corresponding to invalid YAML files, or containing paths that are not supported by the GHA language). To do so, we used the validity flag provided in the dataset, indicating whether a workflow file conforms to the JSON Schema for GHA workflows.[6] A manual inspection revealed that some *a priori* valid workflow snapshots contained *constructs* not supported by the GHA language. We manually examined all unique constructs extracted from the workflows, and compared them with the official GHA documentation [10]. Constructs not mentioned in the documentation were considered invalid, and the 20,750 snapshots containing them were excluded.

After this postprocessing phase, we obtained a final dataset of 2,847,199 workflow snapshots corresponding to 259,661 workflows from 48,952 GitHub repositories, containing 186,706,963 paths.

IV. RQ_0: What are the constructs and features of the GHA language?

Before being able to analyse how GHA is used in practice, we provide a high-level view of the constructs and features that the language provides. As a first step, we used the official GitHub documentation [10] to identify the constructs provided by GHA. Given that this documentation is scattered across multiple webpages, manually collecting all language constructs is challenging [8], so we might have missed some constructs in doing so.

Therefore, as a proxy, we parsed all GHA workflows in our dataset to extract the language constructs. Since our dataset spans multiple years and the GHA language has evolved over time, some detected constructs that existed previously might have been removed from the documentation (e.g., permissions.repository-projects) and new constructs may have been added (e.g., permissions.attestations). Therefore, we only considered the .

[6]https://www.schemastore.org/github-workflow.json

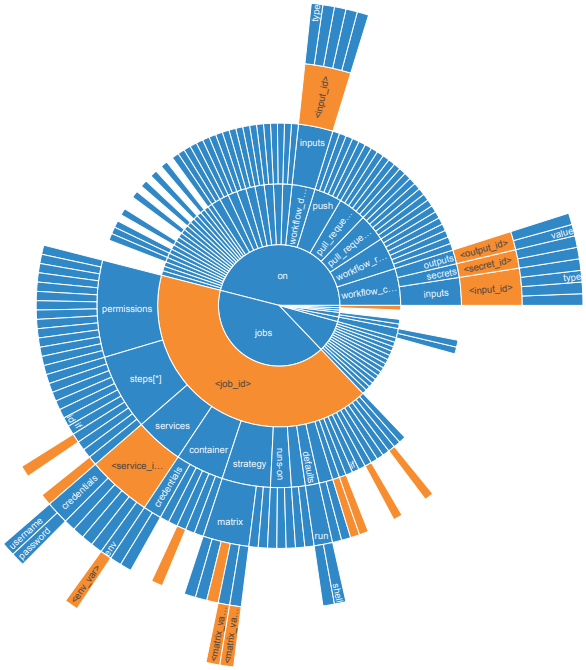


Fig. 2. Sunburst diagram of GHA language constructs. Blue segments are predefined keys in workflows, orange segments are user-defined keys.

This involved 6.5M+ paths from 171,194 workflow snapshots, enabling us to identify 197 valid unique language constructs, of which 119 are workflow-level, 65 are job-level, and 13 are step-level constructs. This set of snapshots is also used to answer RQ_1 and RQ_2.

The sunburst diagram in Figure 2 shows the hierarchical structure of the identified constructs. For readability purposes some construct names were hidden in this figure, but they are available in an interactive version of the diagram in the replication package. One can easily observe that predefined workflow keys (blue segments) are much more abundant than user-defined keys (orange segments). Each segment's size is proportional to the number of sub-keys it contains. The inner ring of the diagram represents top-level keys such as jobs, triggers (on key), workflow-level permissions, and environment variables. The subsequent rings represent constructs at deeper nesting levels. The figure reveals that the GHA language contains lots of constructs at different levels in the hierarchy, especially for the workflow triggers (on) and the individual jobs. This may lead to cognitive load for maintainers to learn to use such constructs effectively. In addition, the many orange segments in Figure 2 reveal that GHA enables workflow maintainers to extend the language with user-defined constructs, such as custom environment variables, parameters, or job names. While this adds flexibility to the language, these variation points add an extra layer of complexity for

TABLE I
LIST OF GHA LANGUAGE FEATURES AND NUMBER OF CONSTRUCTS (197 IN TOTAL) MAPPED TO EACH FEATURE.

Feature	# Constructs	Example Construct
triggers	85	*on.push.branches[∗]*
permissions	30	*permissions . contents*
workflow reuse	14	*jobs.<id>.uses*
job orchestration	12	*jobs.<id>.if*
containers	9	*jobs.<id>.container.image*
matrix strategy	8	*jobs.<id>.strategy . matrix.<var>*
commands	7	*jobs.<id>.steps[∗].run*
services	7	*jobs.<id>.services.<s_id>.image*
environment vars	6	*jobs.<id>.steps[∗].env.<var>*
naming	5	*name*
context	5	*jobs.<id>.runs−on*
Action reuse	3	*jobs.<id>.steps[∗].uses*
step orchestration	3	*jobs.<id>.steps[∗].if*
deployment	3	*jobs.<id>.environment*

maintainers to learn and use the language effectively.

> The rich hierarchically-structured language offered by GHA is a double-edged sword: while providing flexibility and expressiveness to define complex workflows, it may increase the cognitive load for maintainers to learn and use the language effectively.

Construct-level analysis alone is too granular for understanding the broader capabilities of GHA. Therefore, we grouped and mapped all semantically-related constructs into higher-level *features* reflecting workflow capabilities. For example, the constructs *jobs.<id>.steps[∗].run* and *defaults . run . shell* both relate to running commands, so they were mapped to a commands feature encompassing the seven different constructs. Similarly, the construct *jobs.<id>.runs−on* relates to the execution context of a job, so it was mapped to the *context* feature along with four other constructs.

Since GHA does not provide an official taxonomy of such features, we defined our own taxonomy based on a manual analysis of the constructs and the documentation. This process was iteratively conducted by the authors until consensus was reached. The final mapping consists of 197 constructs grouped into 14 features. Table I shows the list of features and the number of constructs mapped to each feature, along with an example construct for each feature. The full mapping is available in the replication package. Table I reveals that the number of constructs per feature ranges from 3 to 85. With 85 constructs, feature *triggers* is the most populated because GHA provides a wide range of events to trigger workflows, each with its own set of constructs to configure the trigger's behavior. The *permissions* feature contains 30 constructs, reflecting the importance of access control in workflows. The ability of GHA workflows to reuse pre-existing workflows or Actions is reflected by the features *workflow reuse* and *Action reuse*, containing 14 and 3 constructs, respectively. Along with *Action reuse*, the features *step orchestration*, and *deployment* also contain only three constructs.

> GitHub Actions offers many features that vary in number of constructs. The cognitive burden on workflow maintainers may depend on the kind of features they need to use. Features that contain more constructs or that are less common may be more difficult to master.

V. RQ_1: WHICH CONSTRUCTS ARE USED IN PRACTICE?

Using the GHA constructs identified in RQ_0, we examine how their usage varies across the same 171,194 workflow snapshots.

#Paths and #Constructs. A distribution analysis of the number of paths and constructs per workflow reveals that workflows typically contain many paths but fewer constructs. For instance, *#Paths* ranges between 2 and 7,712, with a median of 24, while *#Constructs* ranges between 2 and 48, with a median of 11. Half of the workflows have between 15 (Q1) and 42 (Q3) paths and between 9 (Q1) and 14 (Q3) constructs. This already indicates that some constructs tend to be repeated multiple times within workflows.

Inequality in construct usage. To understand how construct usage is distributed across workflows, we computed the Gini coefficient, which is a measure of inequality in a distribution [11]. A Gini coefficient of 0 indicates perfect equality (all constructs are used equally), while a coefficient close to 1 indicates high inequality (a few constructs dominate the usage). We obtained a Gini coefficient of 0.84, indicating strong inequality in construct adoption across workflows. In fact, the 10 most frequently used constructs account for 58.1% of all construct occurrences across our dataset, and the top 50 account for 92.9%. The remaining 147 constructs (74.6%) account for only 7.1%.

Next, we examine how many constructs and paths workflows typically use. A Spearman rank correlation of $\rho = 0.78$ ($p < 0.001$) reveals that the *#Paths* generally increases with *#Constructs*. Still, many workflows have many paths but few constructs, indicating repeated use of the same constructs. As can be observed from the segment sizes in Figure 2, repetition can occur through user-defined keys, multiple jobs, and multiple steps. Overall, workflows vary in diversity (more distinct constructs) and repetition (repeated use of the same constructs).

Most popular constructs. To understand which constructs are most popular in workflows, we analysed the frequency of construct usage across all workflows. Table II reports the top 15 constructs based on workflow usage frequency (%wf) alongside the total number of observed occurrences (# column). The top construct is the non-mandatory workflow *name*, suggesting maintainers are likely to provide a name for their workflow. The second most frequent construct is *runs−on* to specify the runner to execute the job. The overwhelming majority of workflows **not** using this construct instead relied on a reusable workflow (through *jobs.<id>.uses*, not part of the top 15) that was used to specify the runner. Seven of the fifteen top constructs occur inside individual steps (*jobs.<id>.steps [∗]...*). For example, the third and fourth

TABLE II
TOP 15 MOST FREQUENT CONSTRUCTS BASED ON THEIR OCCURRENCES.

	Construct	#	MOW	% WF
1	*name*	169K	—	99.1%
2	*jobs.<id>.runs−on*	281K	1	95.4%
3	*jobs.<id>.steps[∗]. uses*	789K	3	94.2%
4	*jobs.<id>.steps[∗]. with.<param>*	1,195K	5	86.0%
5	*jobs.<id>.steps[∗]. name*	1,168K	5	85.6%
6	*jobs.<id>.steps[∗]. run*	716K	3	77.2%
7	*on.push.branches[∗]*	93K	1	42.2%
8	*jobs.<id>.name*	126K	1	40.1%
9	*jobs.<id>.steps[∗]. env.<var>*	227K	2	31.3%
10	*jobs.<id>.steps[∗]. if*	177K	2	27.4%
11	*jobs.<id>. strategy . matrix.<var>*	97K	1	27.0%
12	*jobs.<id>.steps[∗]. id*	103K	1	26.8%
13	*on. workflow_dispatch*	42K	—	24.5%
14	*on. pull_request . branches[∗]*	48K	1	22.4%
15	*jobs.<id>.if*	55K	1	19.0%

— for MOW signals constructs that can only be used once in a workflow.

top constructs have to do with reusing existing Action components: *uses* specifies the Action to be used, and *with.<param>* provides input parameters to the Action. This reveals that reusing Actions in steps is common practice (94.2% of workflows), much more so than providing custom scripts using the *run* key (ranked 6th, 77.2% of workflows). Constructs related to defining workflow triggers (*on* key) are also quite common. Out of the 34 trigger types in our dataset, *push* (ranked 7th), *workflow_dispatch* to manually trigger a workflow (ranked 13), and *pull_request* (ranked 14) are the most common.

To complement the frequency analysis, we also report the *Median number of Occurrences per Workflow* (MOW) in Table II. It shows not only whether a construct is widely adopted across workflows, but also how often it is used within a single workflow. For example, the fourth and fifth top constructs in Table II have a median of 5 occurrences per workflow. The third top construct has a MOW of 3. This indicates that maintainers typically use multiple Actions on multiple *steps* in one or more *jobs* that requires passing one or more parameters and providing a name for multiple steps. Most of these multiply-used constructs reside at the step level, and suggest that the complexity of workflows can emerge not only from the diversity of constructs used, but also from the repeated use of the same construct within a workflow.

> Most workflows use a small subset of the 197 GHA constructs. Roughly half of them use between 9 and 14 constructs, with a skewed distribution: a few constructs appear in nearly every workflow, whereas many constructs are rarely ever used. The most common constructs relate to using Action components and defining triggers. Workflows also differ in how they use constructs. Some workflows rely on a wide variety of constructs but reuse them less frequently. Others stick to a smaller set of constructs but use them repeatedly.

VI. RQ_2: WHICH FEATURES ARE USED IN PRACTICE?

Based on the grouping of constructs into features (see RQ_0), we analyse how frequently workflows use features, and whether all constructs for a given feature are being used or only a subset of them. Table III summarises three characteristics for each feature across all 171,194 workflow snapshots: the *Usage Rate* (i.e., the proportion of workflows using a feature), the distribution of its *Construct Coverage*, and the distribution of the *Path-to-Construct Ratio*.

Given a feature f and a workflow w, the *Construct Coverage* is computed by counting how many constructs from f are used in w divided by the total number of constructs associated to f. The *Path-to-Construct Ratio* is computed by dividing the number of paths in w by the number of constructs associated to f used in w. Table III caps this ratio at 10 (95.6^{th} percentile) for readability.

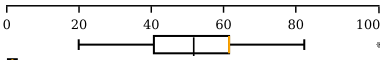
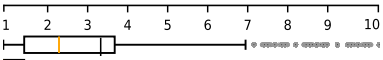
As an example of how to interpret these distributions, consider the *permissions* feature, encompassing 30 constructs (cf. Table I) and used in 32% of all workflows. The *Construct Coverage* boxplot reveals that few of the constructs are used in practice: for instance, half of the workflows (median value, orange line) use no more than 6.7% (i.e., 2 out of 30) *permissions* constructs. The *Path-to-Construct Ratio* boxplot reveals a median of 1, implying that half of the workflows use *permissions* constructs only once (i.e., they define permissions once, at the level of the workflow, job or step, but do not redefine permissions multiple times).

Usage Rate. Table III reveals that some features are more widely adopted than others. Four features are used by more than 94% of workflows. 99.9% of all workflows use *naming*, implying that assigning a name to a workflow, job, or step is a common practice for workflow maintainers. The high usage rate of *triggers* and *context* is due to the necessity to define a trigger and runner for being able to execute a workflow. Finally, the high usage rate of *Action reuse* (94.2%) reflects that reusable Actions are crucial components of most workflows. The alternative way to define steps through custom scripts has a lower usage rate (feature commands, 77.5%), suggesting that the wide availability of reusable Actions on the GitHub Marketplace makes *Action reuse* a preferred practice.

In comparison, the *workflow reuse* feature (12.4%) is approximately eight times less used, suggesting that reusable workflows are either not well-known or too restrictive to be widely useful. Five features are only moderately used: *environment variables* (45.6%), *job orchestration* (38.9%), *permissions* (32.0%), *matrix strategy* (30.7%), and *step orchestration* (28.9%). They reflect more advanced workflow mechanisms that are generally useful to optimise and secure workflows, but less likely to be used by novice workflow maintainers. Finally, three features are rarely used, namely *deployment* (3.8%), *container* (3.1%), and *services* (1.6%). They are considerably more specialised, explaining their low adoption rate. For example, the *services* feature is used for hosting service containers for workflow jobs, which is only needed in specific situations.

Construct Coverage. To analyse how features are used in practice, we examine the distributions of *Construct Coverage* in Table III. The table reveals that not all features are used in the same way. For instance, the boxplots for *naming*,

TABLE III
USAGE STATISTICS OF GHA FEATURES IN PRACTICE. ORANGE LINES IN THE BOXPLOTS REPRESENT THE MEDIAN, BLACK LINE THE AVERAGE.

	Feature	Usage Rate	Construct Coverage	Path-to-Construct Ratio
1	naming	99.9%		
2	triggers	96.3%		
3	context	95.6%		
4	Action reuse	94.2%		
5	commands	77.5%		
6	environment variables	45.6%		
7	job orchestration	38.9%		
8	permissions	32.0%		
9	matrix strategy	30.7%		
10	step orchestration	28.9%		
11	workflow reuse	12.4%		
12	deployment	3.8%		
13	container	3.1%		
14	services	1.6%		

Action reuse, and *services* show a high median construct coverage (60.0%, 66.7%, and 57.1% respectively), implying that workflow maintainers using these features tend to use more constructs associated to them. Since features *naming* and *Action reuse* contain a low number of constructs (5 and 3 respectively), it is easier to use more of their constructs and reach a high construct coverage. The high median construct coverage for the rarely used *services* feature (composed of 7 constructs) suggests that using it in workflows requires more complex configurations.

It is worth noting that the construct coverage distributions of many features are skewed, indicating that while most workflows use only a few constructs, a small number of workflows tend to use a much larger number of constructs. This suggests that some workflows leverage a much larger share of the available constructs, reflecting more advanced configurations.

Path-to-Construct Ratio. This metric captures how many times the constructs of a feature are used repeatedly within a workflow. A ratio of 1 indicates that the constructs of the feature are used only once in the workflow, while a ratio above 1 indicates that the workflow uses some constructs multiple times. Table III reports on the distribution of this metric for all features. We obtain a median ratio of 1 for six features: *context*, *triggers*, *permissions*, *container*, *job orchestration*, and *deployment*. Combined with the results of the previous feature-level metrics analysed, we can conclude that these features tend to be used in a simple way, using only 1 or 2 constructs without repetition.

In contrast, features like *naming*, *matrix strategy*, *step orchestration*, *environment variables*, *workflow reuse*, and *services* have a median value between 1 and 3, and features like *Action reuse* and *commands* have a median value above 3, which shows that some constructs associated to these features are used multiple times within a workflow. Combined with the high median construct coverage for *naming*, *Action reuse*, and *services*, this suggests that these features are used to their full extent (using most of their constructs), and that some of their constructs are used repeatedly in a workflow.

The 14 GHA *features* show varying levels of adoption and usage patterns. Features such as *naming*, *triggers*, *context*, and *Action reuse*, are widely adopted and often used repeatedly within a workflow. Features such as *commands*, *environment variables*, and *job orchestration* are moderately adopted and their constructs may be used multiple times within a workflow. Specialized features such as *workflow reuse*, *deployment*, *container* and *services* are rarely adopted, with the latter two features showing high construct coverage, indicating that they are more complex and may require more advanced configurations.

VII. RQ_3: HOW DOES THE USAGE OF THE GHA LANGUAGE EVOLVE OVER TIME?

While previous sections analysed construct and feature usage on recent workflow snapshots only, this RQ aims to provide insights into how GHA usage has evolved over time. For each month between July 2019 and August 2025, we materialized a snapshot of all workflows that were alive at that time (i.e., created before that date and not yet deleted), resulting in 74 monthly snapshots for 243,008 unique workflows. We excluded 16,653 workflows with a lifespan of less than one month to ensure that we are analyzing workflows that had a chance to evolve over time.

We analysed the evolution of workflow size using two different metrics: *#Paths* and *#Constructs*, aggregated by the mean, median, 25^{th} and 75^{th} percentile. Figures 3-A and 3-B show their monthly evolution. The two figures reveal a clear growth in the *#Paths* and *#Constructs*, indicating a trend towards increasing workflow size. For instance, the median *#Paths* almost doubled over the observation period, from 16 to 31 paths, and the average *#Paths* has even tripled from 20 to 65 paths. In contrast, the median *#Constructs* increased from 9 constructs in August 2019 to 11 constructs in August 2025, suggesting that workflows grow mostly in their number of paths, and much less in the constructs they use.

To confirm these trends, we applied the non-parametric Mann-Kendall test [12] to check for the presence of a mono-

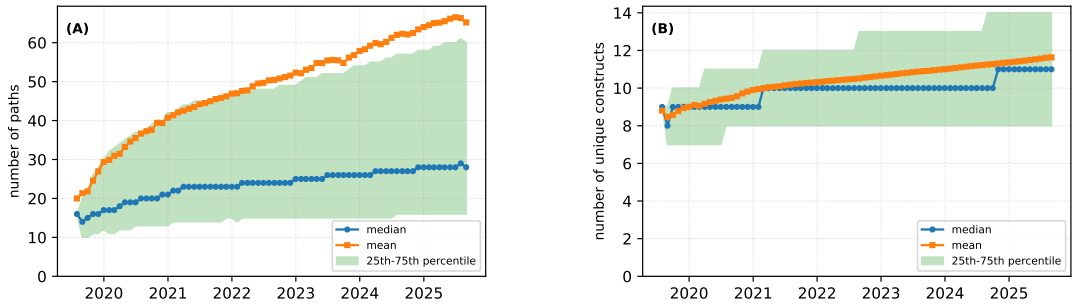


Fig. 3. Monthly evolution of **(A)** *#Paths*, **(B)** *#Constructs*.

tonic trend in the *#Paths* and *#Constructs* metrics over time. The test results indicate a statistically significant increasing trend for both metrics ($p < 0.001$), confirming that the observed growth in *#Paths* ($\tau = 0.95$) and *#Constructs* ($\tau = 0.76$) represents a consistent upward trend over the observation period.

We also analysed the evolution of the *#Features* used in workflows. We found that this metric tends to remain relatively stable over time, with a median of 6 features used in a workflow during the entire observation period (mean ranged from 5.8 to 6.6). This suggests that while workflows are growing in size, they are not necessarily using more features of the GHA language. To get a better understanding of the evolution of feature usage, we analysed the monthly proportion of workflows using each of the 14 features. The results are shown in Figure 4. It reveals that the six most frequently used features have remained largely stable over time and did not change their relative order. Likewise, the three least frequently used features have remained stable over time at a very low median usage rate (never exceeding 4%). Interestingly, the usage rate is consistently increasing over time for three features: *job orchestration*, *permissions*, and *workflow reuse*. The last two were not part of the GHA language when it was first introduced in 2019 but were made available in 2021, illustrating the evolution of GHA language usage by the introduction and take-up of new language features, following GitHub's efforts for security hardening and DRY practices. One can also observe that since the first appearance in our dataset of *workflow reuse* (October 2021), the usage of some features such as *context*, *Action reuse* or *commands* has begun to decrease slightly. This suggests a substitution effect where workflow maintainers are more and more resorting to reusable workflows to replace the functionalities previously implemented using existing features.

> Workflow size has increased during the observation period by a factor of two to three in terms of number of paths, but only slightly in terms of number of constructs. There is no observable growth in the number of features used, but specific features are seeing an increase in usage rate, in line with the introduction and promotion of new features in the GHA language.

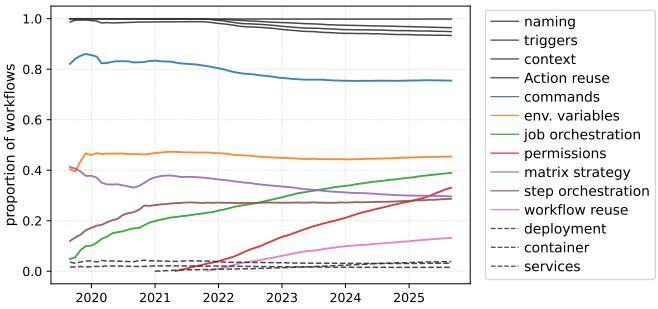


Fig. 4. Monthly evolution of *Feature Usage* in GHA workflows.

VIII. RQ_4: To what extent does workflow size impact reliability and maintainability?

Previous RQs focused on quantifying and analysing the evolution of workflow size in terms of *#Paths*, *#Constructs*, *#Features*, and *Path-to-Construct-Ratio*. Empirical analyses of traditional source code artefacts (e.g., software programs) have used and studied a wide range of code size metrics, such as the number of lines of code, the number of tokens, and the number of syntactic elements (e.g., functions, classes, packages) [13]. Such size metrics have been established as strong, valid predictors of external quality features, specifically bug-proneness and maintenance effort [14], [15].

Drawing on this analogy, we hypothesise workflow size to be related to reliability and maintainability issues. Our intuition is that larger workflows are more difficult to understand, test, and maintain, potentially leading to more frequent execution failures and higher maintenance effort. To verify this hypothesis, we selected several metrics, used to measure *maintainability* and *reliability* of traditional software artifacts, and applied them to GHA workflows:

Failure Rate is the percentage of workflow executions that fail. This metric is commonly used to measure the reliability of software artifacts [16].

#Commits is the number of commits made to a workflow during a specific time window. This metric has already been used to measure the maintenance effort of GHA workflows [17], [18]. The use of a fixed, common time interval avoids biasing towards workflows that existed for a longer time, hence having more opportunity to be changed through commits. It

TABLE IV
MANN-WHITNEY U COMPARISON OF RELIABILITY AND MAINTAINABILITY METRICS BETWEEN SMALL (FIRST THIRD) AND LARGE (LAST THIRD) WORKFLOWS.

	workflow size		Cliff's δ (***medium***, **small** or negligible)			
Size metric	small $\leq$	large $>$	***Failure R.***	***#Commits***	***Avail.***	***MTTR***
#Paths	29	61	**0.228**	***0.374***	**−0.247**	**0.174**
#Constructs	12	16	**0.151**	***0.411***	**−0.175**	**0.186**
#Features	7	8	**0.172**	***0.332***	**−0.190**	0.135
Path-to-Constr.	2.31	4.02	**0.236**	**0.300**	**−0.250**	0.107

also avoids possible bias due to major changes that may have occurred over time in the language, technology or practices.

Median Time to Repair (MTTR) measures the median time required to restore a workflow to a working state after a failure. This is a standard measure of maintainability of software artifacts [16]. A long time to repair can indicate that the workflow is more difficult to maintain. To avoid bias towards repeated failures within the same workflow, we compute *MTTR* starting from the first observed failure of each workflow during a specific time window. We also exclude workflows that never recovered from the failure during this period.

Availability (a.k.a. uptime) is the proportion of time that a workflow is not in a failed state [16].

To compute these metrics, we retrieved the execution results of the commits made to the workflows over a six-month period from 20 January to 18 July 2025. For each commit made to a workflow during this period, we used the GitHub REST API to collect the results of the run that were triggered by a change to the workflow file (either through a push or a pull request). By only considering commits that modify the workflow file contents and by focusing on the runs that were triggered by these changes, we reduce the effect of external factors that could affect the outcome of workflow execution (e.g., changes in the code base or external environment). This resulted in 74,438 workflow run results associated with 13,915 workflows.

To understand the relationship between workflow size and reliability/maintainability, we used each of the four workflow size metrics to split all these workflows into three balanced groups reflecting **small**, **medium**, and **large** workflows. For example, based on the *#Paths* metric, the **small** group contained 4,730 workflows with $\leq$29 paths, the **medium** group contained 4,470 workflows with between 29 and 61 paths, and the **large** group contained 4,715 workflows with $>$61 paths. The discrete nature of the metric resulted in a slight variation in the number of workflows contained in each group.

To assess the effect of each size metric on the four reliability and maintainability metrics, we performed 16 non-parametric Mann-Whitney U tests to compare between the **small** and **large** groups. For *MTTR*, we only considered 2,691 workflows as it is computed only on workflows that successfully recovered from an initial failure. We applied a Benjamini-Hochberg adjustment to control the false discovery rate due to multiple hypotheses testing [19], and set a significance level threshold of α=0.01. We could reject all null hypotheses with statistical significance: **large** workflows tend to have a higher *Failure Rate*, more *#Commits*, lower *Availability*, and a higher *MTTR* than **small** workflows.

Table IV reports on the effect size of the differences, computed using Cliff's δ and interpreted based on Romano et al. [20] (i.e., *negligible* effect if $|\delta| < 0.147$, *small* if $0.147 < |\delta| < 0.33$ and *medium* effect if $0.33 \leq |\delta| < 0.474$). One can observe *medium* effect for 3 comparisons, *small* effect for 11, and *negligible* effect for only 2 *MTTR* comparisons. These results suggest that larger workflows are more likely to fail, require more maintenance effort, and have lower availability.

While the Mann-Whitney U tests establish significant differences between **small** and **large** workflow groups, to gain deeper knowledge on the link between the increase of size metrics and workflow reliability and maintainability, we rely on Generalized Linear Modeling (GLM) [21]. It extends linear regression models via a link function to support response variables with a non-normal distribution. We excluded workflows with $<$ 3 runs, to avoid bias by workflows with very few runs. Since *Failure Rate* is bounded between 0 and 1, we use a binomial logistic regression, which is well-suited for this type of data. Its effect size is reported using *Odds Ratio* (OR). For the discrete *#Commits* metric, we use a negative binomial regression, which is suitable for overdispersed count data. Its effect size is reported using *Incidence Rate Ratio* (IRR). *Availability* and *MTTR* could not be included in the GLM analysis, since their data distribution characteristics (e.g., heavy right-skewness for *MTTR* and near-constant values for *Availability*) do not make them amenable to GLM.

Table V reports the effect sizes of the regression analyses (OR for *Failure Rate*, IRR for *#Commits*), along with their 95% confidence intervals. All regression analyses were found to be statistically significant (α=0.01) after Benjamini-Hochberg adjustment [19]. Effect sizes (OR and IRR) were all $>$1, indicating that an increase in the predictor variable is associated with an increase in the outcome variable. Predictor variable *#Features* had the largest effect on both outcome variables. Adding one feature to a workflow is associated with an 18.9% increase in the odds of failure and an 8.5% increase in *#Commits*. The second best predictor variable was *Path-to-Construct Ratio*, with an increase of one unit being associated with a 13% increase in the odds of failure and a 7% increase in *#Commits*. The predictive power of *#Constructs* and *#Paths* was considerably smaller.

This suggests that workflow maintainers should be cautious when creating or modifying large workflows, as they are more likely to face reliability and maintainability issues. This aligns with evidence from programming language research that suggests, for example, to keep method sizes small, since large methods are associated with more bug-proneness and lower maintainability [15]. In a CI/CD environment where workflows execute frequently and are notoriously difficult to test and debug locally, even a 5% cumulative increase in failure odds may result in significant wasted computing resources and manual troubleshooting efforts.

TABLE V
EFFECT SIZES AND CONFIDENCE INTERVALS OF GLM REGRESSION OF WORKFLOW SIZE METRICS ON *Failure Rate* AND *#Commits*.

	Failure Rate		*#Commits*	
Size metric	**OR**	**95% CI**	**IRR**	**95% CI**
#Paths	1.005	1.004–1.005	1.004	1.003–1.004
#Constructs	1.024	1.018–1.029	1.048	1.042–1.054
#Features	1.189	1.171–1.208	1.085	1.068–1.102
Path-to-Construct Ratio	1.130	1.121–1.138	1.073	1.063–1.083

> There is a statistically significant relation between workflow size and reliability and maintainability, with the number of unique features being the strongest predictor. Larger workflows are more likely to have more failures, more commits, longer downtime (i.e., lower availability), and take more time to repair (i.e., higher *MTTR*).

During our personal experience with maintaining workflows, we observed some features to be more error-prone to use and maintain than others. For example, custom *commands* in steps often require quite some technical expertise that makes them difficult to test and debug, whereas the alternative of relying on pre-built reusable Actions that have been tested by the community tends to reduce the likelihood of errors and failures. To verify and generalise this hypothesis, we repeated the GLM regression analysis at the level of individual features, in order to determine which of these features are more likely to be associated with reliability and maintainability issues. To ensure reliable estimations of effect size, we only focus on features that are used by more than 5% and less than 95% of the 13,915 considered workflows. Therefore, the universal features *naming* and *triggers* (used by more than 95%), and the highly specialized features *services* and *deployment* (used by fewer than 5%) are excluded from this analysis.

Table VI reports two regression analyses to relate workflow features to both *Failure Rate* and *#Commits* of workflows. The first analysis considers the presence (i.e., the use) of a given feature in a workflow. The second analysis considers a count variable indicating the number of paths added for a given feature in a workflow. For example, workflows using shell *commands* have over four times higher odds of failing compared to those that do not (OR=4.12) and are associated with a 15% reduction in *#Commits* (IRR = 0.85). Adding additional command paths incrementally increases maintenance effort (IRR = 1.019 per path). The presence of an *environment variable* in a workflow increases the odds of failure by 84% and increases the *#Commits* by 35%, suggesting that using environment variables can be error-prone and require more maintenance effort. Adding one more environment variable increases the odds of failure and the *#Commits* by 3% each. The presence of the *step orchestration* feature in a workflow increases the odds of failure by 79% and the *#Commits* by 42%, while adding one more path to this feature increases the odds of failure and the *#Commits* by 2% each. Interestingly, the presence of *Action reuse* in a workflow decreases the odds of failure by 28% (OR=0.72) but increases maintenance effort by 58% (IRR = 1.58). This suggests that using *Actions* instead of shell commands in workflows could be a more reliable option, although it comes with additional maintenance effort to keep these Actions up-to-date.

TABLE VI
EFFECT OF FEATURE PRESENCE AND FEATURE PATH COUNT ON *Failure Rate* AND *#Commits* FROM GLM REGRESSION.

	Feature Presence		**#Paths for the feature**	
	Failure Rate	*#Commits*	*Failure Rate*	*#Commits*
Feature	**OR**	**IRR**	**OR**	**IRR**
Commands	4.12	0.85	1.018	1.019
Environment variables	1.84	1.35	1.032	1.031
Step orchestration	1.79	1.42	1.023	1.025
Job orchestration	1.66	1.20	1.023	1.047
Matrix strategy	1.46	1.16	1.016	1.013
Action reuse	0.72	1.58	1.012	1.013
Permissions	0.70	1.42	0.838	1.086
Context	—	1.58	1.099	1.064
Container	1.58	1.32	—	0.391
Workflow reuse	1.49	1.17	—	—

All results except those marked as — are statistically significant. The 95% confidence intervals are available in the replication package.

> Relying on custom shell *commands* significantly increases the odds of workflow failure, whereas adopting *reusable Actions* reduces the rate of failure but significantly increases ongoing maintenance effort (*#Commits*) to manage external dependencies.

IX. RELATED WORK

Many empirical studies have investigated the adoption, evolution, and challenges of CI/CD pipelines in open-source projects, primarily focusing on Travis CI. These works highlighted the benefits of rapid feedback and pain points such as configuration bad smells, complex job matrices, and long build times [1], [22]–[26].

Soon after its introduction, GHA became the dominant CI/CD service due to its deep integration with GitHub and a large marketplace of reusable Actions [2]. Studies have shown that GHA workflows are widely adopted but can become structurally complex [4]. They require continuous maintenance driven by bug fixes and CI/CD improvements [18]. Expanding on this, empirical analyses have revealed that GHA workflows are frequently modified, particularly in task specifications and configurations [27]. This continuous evolution provides evidence that GHA workflows follow Lehman's laws of continuing growth and continuing change [28].

Despite their proven benefits, maintaining reliable GHA workflows is challenging. Saroar and Nayebi [5] reported that YAML configurations are error-prone, while Zheng et al. [8] found that frequent workflow failures waste computing resources and require significant maintenance effort. Looking at build outcomes specifically, Huang and Lin [29] investigated GHA workflow reruns caused by flakiness. They used structural metrics (e.g., lines of code and job count) alongside execution history as features for machine learning models to predict execution outcomes. While their work focuses on

predicting immediate rerun success, our study provides a fine-grained analysis of how specific usage of GHA language and size metrics impact workflow reliability and maintainability.

CI/CD configuration size directly impacts workflow maintainability. Using a basic size metric (the number of directives) as a proxy for configuration complexity, Ghaleb et al. [7] found GHA to be the second most complex CI/CD service. In software engineering research, size and complexity metrics are well-established predictors of maintainability and defect-proneness. Gil and Lalouche [14] demonstrated that the validity of many code complexity metrics can be attributed to the confounding effect of size, while Chowdhury et al. [15] provided empirical evidence linking size directly to future maintenance effort and bug-proneness. Despite these insights, there is a lack of comprehensive understanding regarding how specific GHA language size and complexity metrics impact workflow reliability and maintenance effort. Our study bridged this gap by investigating the relationship between multiple GHA workflow size metrics and workflow reliability and maintainability.

X. Threats to Validity

To discuss the threats to validity of our research we follow the structure recommended by Wohlin et al. [30].

Construct validity concerns the relation between theory and observed findings. The dataset we relied on [9] includes only valid YAML workflow files that conform to the GHA JSON schema. Still, some included workflows may have been disabled in practice. We excluded workflows that use unsupported constructs (see Section III). Our construct catalog was derived from workflows alive as of August 2025, and may not cover constructs absent from all considered workflows. Given the scale of the dataset, however, the probability of missing a widely used construct is very low. We are therefore confident that our results reflect representative usage patterns. Finally, the execution results collected for RQ_4 pertain to commits that modified workflows. We cannot claim that failing workflow runs can solely be attributed to changes made to the workflow itself. External factors (e.g., dependency updates, third-party service outages) may also contribute.

Internal validity concerns factors that could influence observed outcomes independently of the intended measurements. In Section IV we manually mapped constructs to features. To mitigate bias, the authors independently proposed features and assigned constructs to them based on the official GHA documentation. Disagreements were then resolved by consensus.

External validity concerns the generalisability of our findings. Our dataset is restricted to public GitHub repositories with at least 100 stars and 300 commits, criteria commonly used to exclude abandoned or experimental projects [31]. Consequently, our findings may not generalise to smaller, less active repositories, nor to workflows hosted in private repositories.

XI. Discussion and Conclusion

Although the GHA language defines 197 constructs across 14 features, most workflows rely on only a small subset of them, typically between 9 and 14 constructs. Features such as *triggers*, *naming*, *context*, and *Action reuse* dominate everyday practice, while more specialized features like *containers*, *services*, and *workflow reuse* remain rare and tend to demand more complex configurations when adopted. This imbalanced situation suggests that GHA usage comes with a steep learning curve, and that the cognitive burden on workflow maintainers scales with the breadth and rarity of the features they need to employ. This raises questions about whether these features are niche by nature or whether their documentation and tooling support are insufficient.

Analysing the evolution of 243K workflows over a six-year period showed that workflow size doubled in terms of paths, though the number of constructs per workflow increased only slightly. We also found that newly introduced or promoted features (such as *permissions* or *workflow reuse*) are gradually gaining adoption, reflecting the evolving nature of the language and its platform. This highlights the need for maintainers to continuously keep pace with language changes.

Critically, workflow size has measurable relationship with reliability and maintainability. Our results indicate that larger workflows, especially those with more paths and a higher path-to-construct ratio, are associated with a higher failure rate, a lower availability, and a longer recovery time. We also found that specific features tend to be associated with such reliability and maintainability risks. For instance, using *commands* and *environment variables* is associated with a higher failure rate, while using *reusable Actions* is associated with a lower failure rate but a higher maintenance effort. These findings suggest that certain language constructs may introduce more complexity or be more difficult to use correctly, leading to increased maintenance challenges and reliability issues.

Based on these insights, we outline actionable insights for different stakeholders. Workflow maintainers should be aware that larger workflows, especially those using a wider variety of features and constructs, are more failure-prone and require more maintenance effort. Understanding which features and constructs are more failure-prone can help practitioners make informed decisions about how to design their workflows to minimize reliability risks and maintenance effort. While we cannot establish causality from our analysis, since there are many additional factors that could influence workflow reliability and maintainability, we still recommend practitioners to limit workflow size, potentially by modularizing their workflows through *workflow reuse* mechanisms.

Tool builders can also benefit from our findings. For instance, linters could warn about overly long workflows and the use of specific features that are associated with higher failure rates or maintenance effort. IDEs could provide suggestions to refactor large workflows, e.g., to replace complex custom *commands* with *Actions* or to move and factorise long step sequences into reusable workflows.

Finally, researchers can use our results as a starting point for more in-depth quantitative and qualitative studies to address unanswered questions on the usage and complexity of GHA. For instance, why are some features more failure-prone or

require more maintenance effort than others? What are the "atoms of confusion" [32] in workflows? Which best practices for using specific features and constructs could help reduce maintenance effort and reliability risks?

XII. Data Availability

The replication package including all scripts, the construct to feature mapping, and instructions to reproduce the analysis is accessible on https://figshare.com/s/224b0b49967f7ea8283a.

References

[1] F. Zampetti, S. Geremia, G. Bavota, and M. Di Penta, "CI/CD pipelines evolution and restructuring: A qualitative and quantitative study," in *Int'l Conf. Software Maintenance and Evolution (ICSME)*, 2021.

[2] M. Golzadeh, A. Decan, and T. Mens, "On the rise and fall of CI services in GitHub," in *Int'l Conf. Software Analysis, Evolution and Reengineering*. IEEE, 2022.

[3] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys*, vol. 37, no. 4, 2005.

[4] A. Decan, T. Mens, P. Rostami Mazrae, and M. Golzadeh, "On the use of GitHub Actions in software development repositories," in *Int'l Conf. Software Maintenance and Evolution*. IEEE, 2022.

[5] S. G. Saroar and M. Nayebi, "Developers' perception of GitHub Actions: A survey analysis," in *Int'l Conf. Evaluation and Assessment in Software Engineering*, 2023.

[6] H. Onsori Delicheh, G. Cardoen, A. Decan, and T. Mens, "Automation and reuse practices in github actions workflows: A practitioner's perspective," *arXiv preprint*, 2026.

[7] T. Ghaleb, O. Abduljalil, and S. Hassan, "CI/CD configuration practices in open-source Android apps: An empirical study," *ACM Trans. Softw. Eng. Methodol.*, May 2025.

[8] L. Zheng, S. Li, X. Huang, J. Huang, B. Lin, J. Chen, and J. Xuan, "Why do GitHub Actions workflows fail? An empirical study," *ACM Trans. Softw. Eng. Methodol.*, Jul. 2025.

[9] G. Cardoen, T. Mens, and A. Decan, "A dataset of GitHub Actions workflow histories," in *Int'l Conf. Mining Software Repositories*. ACM, 2024, pp. 677–681.

[10] GitHub, "Workflow syntax for GitHub Actions," https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions, 2025, accessed: 2025-09-22.

[11] C. Gini, *Variabilità e mutabilità*. Tipogr. di P. Cuppini, 1912.

[12] H. B. Mann, "Nonparametric tests against trend," *Econometrica*, vol. 13, no. 3, pp. 245–259, 1945. [Online]. Available: http://www.jstor.org/stable/1907187

[13] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach, Third Edition*, 10 2014.

[14] Y. Gil and G. Lalouche, "On the correlation between size and metric validity," *Empirical Software Engineering*, vol. 22, pp. 1–27, 10 2017.

[15] S. A. Chowdhury, G. Uddin, and R. Holmes, "An empirical study on maintainable method size in Java," in *Int'l Conf. Mining Software Repositories (MSR)*. ACM, 2022, pp. 252–264.

[16] IEEE, "IEEE guide for the use of IEEE standard dictionary of measures to produce reliable software," New York, NY, USA, 1988.

[17] N. Chopra and T. A. Ghaleb, "From first use to final commit: Studying the evolution of multi-CI service adoption," in *Int'l Conf. Software Maintenance and Evolution (ICSME)*. IEEE, 2025, pp. 773–778.

[18] P. Valenzuela-Toledo and A. Bergel, "Evolution of GitHub Action workflows," in *Int'l Conf. Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022.

[19] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.

[20] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices?" in *Annual Meeting of the Southern Association for Institutional Research*, 2006.

[21] J. A. Nelder and R. W. M. Wedderburn, "Generalized linear models," *Journal of the Royal Statistical Society. Series A (General)*, vol. 135, no. 3, pp. 370–384, 1972.

[22] D. G. Widder, M. Hilton, C. Kästner, and B. Vasilescu, "A conceptual replication of continuous integration pain points in the context of Travis CI," in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019.

[23] K. Gallaba and S. McIntosh, "Use and Misuse of Continuous Integration Features: An Empirical Study of Projects that (mis)use Travis CI," *IEEE Transactions on Software Engineering*, vol. 46, no. 1, pp. 33–50, 2020.

[24] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. Gall, and M. Di Penta, "An empirical characterization of bad practices in continuous integration," *Empirical Software Engineering*, vol. 25, 2020.

[25] O. Elazhary, C. Werner, Z. S. Li, D. Lowlind, N. A. Ernst, and M.-A. Storey, "Uncovering the benefits and challenges of continuous integration practices," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2570–2583, Jul. 2022.

[26] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Int'l Conf. Automated Software Engineering (ASE)*. IEEE, 2016, pp. 426–437.

[27] P. Rostami Mazrae, A. Decan, T. Mens, and M. Wessel, "An empirical study of the evolution of GitHub Actions workflows," *Journal of Systems and Software*, vol. 236, 2026.

[28] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution—the nineties view," in *Proceedings of the IEEE Metrics Symposium (Metrics '99)*, 1999, fEAST/1 Technical Report (public PDF).

[29] J. Huang and B. Lin, "On the reruns of GitHub Actions workflows," *ACM Trans. Softw. Eng. Methodol.*, Feb. 2026.

[30] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer, 2012.

[31] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining GitHub," *Empirical Software Engineering*, vol. 21, no. 5, 2016.

[32] D. Gopstein, H. H. Zhou, P. Frankl, and J. Cappos, "Prevalence of confusing code in software projects: atoms of confusion in the wild," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 281–291. [Online]. Available: https://doi.org/10.1145/3196398.3196432