On the Use of GitHub Actions in Software Development Repositories

Alexandre Decan
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

Pooya Rostami Mazrae
Software Engineering Lab
University of Mons
Mons, Belgium
pooya.rostamimazrae@umons.ac.be

Mehdi Golzadeh
Software Engineering Lab
University of Mons
Mons, Belgium
mehdi.golzadeh@umons.ac.be

***Abstract*—GitHub Actions was introduced in 2019 and constitutes an integrated alternative to CI/CD services for GitHub repositories. The deep integration with GitHub allows repositories to easily automate software development workflows. This paper empirically studies the use of GitHub Actions on a dataset comprising 68K repositories on GitHub, of which 43.9% are using GitHub Actions workflows. We analyse which workflows are automated and identify the most frequent automation practices. We show that reuse of actions is a common practice, even if this reuse is concentrated in a limited number of actions. We study which actions are most frequently used and how workflows refer to them. Furthermore, we discuss the related security and versioning aspects. As such, we provide an overview of the use of GitHub Actions, constituting a necessary first step towards a better understanding of this emerging ecosystem and its implications on collaborative software development in the GitHub social coding platform.**

***Index Terms*—GitHub Actions, continuous integration, collaborative software development, workflow automation**

I. Introduction

Open source software (OSS) development is a continuous, highly distributed and collaborative endeavour [1]. Development of OSS projects faces many socio-technical challenges [2]–[4]. The multitude of tools (e.g., version control systems, software distribution managers, bug and issue trackers) and development-related activities makes it very challenging for contributor communities to keep up with the rapid pace of producing and maintaining high-quality software releases.

Automated workflows were introduced to automate numerous repetitive social or technical activities that are inherently part of the collaborative software development process. Continuous integration, deployment and delivery (CI/CD) have become the cornerstone of collaborative software development and DevOps practices. Well-known examples of CI/CD services are Travis, Jenkins, CircleCI and TeamCity. They automate the integration of code changes from multiple contributors into a central repository where automated builds, tests and code quality checks run.

GitHub is by far the largest social coding platform, hosting the development history of millions of collaborative software repositories, and accommodating over 73 million users in 2021 [5]. GitHub publicly announced the beta version of GitHub Actions (abbreviated to GHA in the remainder of this paper) in October 2018 based on popular demand, and in response to GitLab's integrated CI/CD support [6]. In August 2019, GitHub officially began supporting CI through GHA, and the product was released publicly in November 2019.

GHA [7] allows the automation of a wide range of tasks based on a variety of triggers such as commits, issues, pull requests, comments, schedules, and many more. Its deep integration into GitHub implies that GHA can be used not only for executing test suites or deploying new releases as in traditional CI/CD services, but also to facilitate code reviews, communication, dependency and security monitoring and management, etc. GHA also promotes the use and sharing of reusable components, called *actions*, in workflows. These actions are distributed in public repositories and on the GitHub Marketplace. They allow workflow developers to easily integrate specific tasks (e.g., set up a specific programming language environment, publish a release on a package registry, run tests and check code quality) without having to write the corresponding code.

Since its public release in November 2019, GHA has become the most dominant CI/CD service, only 18 months after its introduction [8]. Its Marketplace of reusable actions has been growing exponentially ever since, reaching 12K reusable actions in February 2022. It is therefore fair to say that GHA has become a software ecosystem of its own, comparable to ecosystems of reusable software libraries (such as npm, RubyGems, CRAN, Maven, and PyPI) that have been empirically studied by many researchers in recent years (e.g., [9]–[14]).

The emerging GHA ecosystem is worthy of being empirically studied in its own right since it is likely to suffer from the same issues related to dependency management, security vulnerabilities, outdated or obsolete components, backward compatibility, and so on. This article therefore quantitatively studies the use of GHA in 68K repositories on GitHub. We analyse which workflows are automated and identify the most frequent automation practices. We show that reuse of actions is a common practice and identify which actions are reused and how. As such, we provide an overview of the use of GHA, a necessary first step towards a better understanding of the emerging GHA ecosystem and its implications on software development in GitHub repositories. More concretely, we answer the following research questions:

- What are the characteristics of GitHub repositories using GHA workflows?
- Which kinds of workflows are automated?
- What are the most frequent jobs in workflows?
- What are the automation practices?
- Which actions are reused and how?
- Which versioning practices are being used?

The remainder of this paper is structured as follows. Section II presents related work. Section III introduces some of the core concepts of GHA and the data extraction process. Sections IV to IX address the research questions. Section X discusses the findings and their implications. Section XI presents the threats to validity of the research, and Section XII concludes.

II. Related Work

Continuous integration has been introduced by Fowler and Foemmel in their seminal blog in 2000 [15]. They outlined 10 core CI practices aiming at increasing the speed of software development and improving software quality. Among others, they stressed fully automated and reproducible builds and tests that run several times a day. Elaszhary et al. [16] discuss the benefits and challenges of these practices in three software-producing companies. They found that these practices are broadly implemented but how they are implemented varies depending on their perceived benefits, the context of the project, and the CI/CD tools used by the organization. They call for more research to understand these differences and how they impact software development and quality. Hilton et al. [17] report on a mixed-methods study to analyse the usage, costs and benefits of continuous integration in open-source projects. They found that CI/CD is widely adopted by the most popular projects and that CI/CD usage continues to grow. They also provide evidence that CI/CD helps projects release more often. Vasilescu et al. [18] found quantitative evidence that usage of certain CI/CD services improves team productivity in GitHub projects, allowing teams to integrate more external contributions without a decrease in code quality. Lamba et al. [19] studied the spread of CI/CD and quality assurance tools in npm package repositories. They notably show that repositories tend to stick to a given CI/CD tool once they adopt it. In addition, by investigating differences in characteristics between early and late adopters, they found that social factors play a significant role in tool adoption. Recently, Soares et al. [20] conducted a systematic literature review on the impact of CI/CD on software development. By analysing empirical evidence from 101 papers ranging from 2003 to 2019, they found that existing research mostly revealed the positive effects of CI/CD on software development, and that further studies are necessary to better understand the trade-offs between adopting CI/CD and overcoming its inherent challenges.

There is a plethora of studies focusing on the use of Travis in GitHub projects [21]–[29]. Vasilescu et al. [21] empirically explored the use of Travis in 918 GitHub projects. They found that projects using Travis have more pull requests accepted, merged, and rejected without a decrease in quality measured in terms of reported bugs. Cassee et al. [23] investigated the impact of Travis on the social aspects of software development, focusing on the code review practices in 685 GitHub projects. Their results show that projects using Travis tend to have fewer discussions in their pull requests, suggesting that developers perform the same amount of work with less communication after the adoption of Travis. Beller et al. [26] analysed failures in over 2,5 million code builds on Travis. They found that testing is the single most important reason why CI/CD builds fail. They also found that the use of multiple integration environments leads to 10% more failures being caught at build time. Widder et al. [29] conducted a mixed-methods study to identify the reasons why projects decided to abandon the use of Travis. By analysing a thousand projects that stopped using Travis between 2011 and 2017, they found that long build times, CI/CD consistency across projects, lack of tests and difficulty to troubleshoot a build failure are among the most frequent reasons to abandon Travis. Zampetti et al. [22] performed a fine-grained mixed-methods study on the evolution of specific Travis pipelines. They analysed 615 CI/CD pipeline configuration change commits, and proposed 16 different metrics to capture how Travis pipelines evolve and get restructured over time.

Golzadeh et al. [8] conducted a quantitative study aiming to better understand the rapidly evolving CI/CD landscape on GitHub. By analysing the use of 20 different CI/CD services from 2012 to 2021 in 91K+ active npm repositories, they observed that GHA has become the most dominant CI/CD service only 18 months after its introduction. They also found that the introduction of GHA coincides with a decreasing adoption rate and an increasing discontinuation rate for other CI/CD services, especially for Travis. While this study showed that traditional CI/CD services on GitHub are getting replaced rapidly by GHA, only very few research articles have studied GHA itself. Kinsman et al. [30] quantitatively analysed the impact of adopting GHA in 3,190 repositories. Their results indicate that the adoption of GHA increases the number of rejected pull requests and decreases the number of commits in merged pull requests. By manually inspecting 209 issues related to GHA, they concluded that developers have a positive perception of GHA. Valenzuela-Toledo and Bergel [31] investigated the use and maintenance of GHA workflows in 10 popular GitHub repositories. They manually inspected 222 commits related to workflow changes and determined 11 different types of workflow modifications. They also uncovered a number of deficiencies in GHA workflow production and maintenance, and call for adequate tooling to support creating, editing, refactoring, and debugging GHA workflow files.

III. Methodology

A. About GHA

To enable GHA on a repository, one has to create one or more YAML files, each describing a single GHA *workflow*, and store them in the `.github/workflows` folder. Fig. 1

```
name: Example of a workflow file
on:
  push:
  pull_request:
  schedule:
    - cron: "0 6 * * 1"
jobs:
  test:
    name: Test project
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: 3.9
      - name: Install dependency
        run: pip install pytest
      - name: Execute tests
        run: pytest
```

Fig. 1. Example of a GHA workflow file.

shows an example of a GHA workflow file.[1] A workflow defines the set of *events* (e.g., a push, a pull request, or a scheduled event) that trigger the execution of a set of *jobs*. A job may reuse an existing workflow through the `uses:` key. Alternatively, a job defines a list of *steps* that will be sequentially executed. Steps are the smallest units of work in a workflow. A step can specify through the `run:` key the commands that will be executed (e.g., `pip install pytest`) or it can delegate its task by calling a predefined *action* or Dockerfile through the `uses:` key (e.g., `uses: actions/setup-python@v2` or `uses: docker://alpine:3.8`). An action is an individual task that can be shared for reuse on a public GitHub repository and on the GitHub Marketplace.[2] Developers can use actions in steps to avoid having to write explicitly the various commands that need to be executed. Actions can access the GitHub API to interact with repositories (e.g., to create a comment in a pull request for test reports), or any third-party API (e.g., to deploy a new release on PyPI). To obtain more information about GHA, the interested reader is invited to consult the official GitHub documentation[3] or Chandrasekara and Herath's book [7].

B. Data Extraction

To conduct an empirical study on the use of GHA in software development repositories, we need a large collection of GitHub repositories. The dataset should exclude repositories that are used only for experimental or personal reasons, or that show no or little traces of actual software development activity [32]. We relied on the SEART GitHub search engine [33] to obtain a list of candidate repositories. We queried the tool on 2022-01-24 to get all non-fork repositories that were created before 2021, which were still active in 2021, and had at least 100 commits and 100 stars. We obtained 67,870 repositories satisfying these criteria.

On 2022-01-24 we locally cloned these repositories to look for the presence of YAML files in the `.github/workflows` folder of their default branch. We parsed these files to check whether they define a GHA workflow, and if applicable, we extracted the relevant data about the workflow (e.g., name, events), about the jobs configured in the workflow (e.g., name, `uses:` key, steps) and about the steps defined in these jobs (e.g., name, commands, `uses:` key). At the end of this process, our dataset covers 67,870 repositories containing 70,278 workflows, 108,500 jobs, and 567,352 steps. The remainder of this article presents what we found in these repositories, workflows, jobs, and steps (including their actions). The data and code to replicate the analysis are available on doi.org/10.5281/zenodo.6634682.

IV. What Are the Characteristics of GitHub Repositories Using GHA Workflows?

Not all GitHub repositories make use of GHA. Out of the 67,870 repositories contained in the dataset, 29,778 of them (i.e., 43.9%) contain at least one workflow file. Table I reports on the number and proportion of repositories having a workflow file, distinguishing repositories based on their main programming language (as specified in the repository metadata on GitHub).

TABLE I
Number and proportion of repositories using GHA workflows, grouped by main programming language.

language	repositories #	repositories %	using GHA workflows % language	using GHA workflows % repo.
JavaScript	13,542	19.6%	34.9%	15.9%
Python	12,319	17.8%	45.9%	19.0%
TypeScript	6,362	9.2%	58.5%	12.5%
Java	6,105	8.8%	39.2%	8.0%
C++	5,701	8.2%	40.9%	7.8%
Go	4,988	7.2%	57.2%	9.6%
C	4,314	6.2%	36.1%	5.2%
PHP	4,005	5.8%	48.2%	6.5%
C#	3,630	5.3%	34.6%	4.2%
Ruby	2,599	3.8%	50.8%	4.4%
Shell	2,327	3.4%	33.2%	2.6%
Swift	1,411	2.4%	34.4%	1.6%
Kotlin	1,150	1.7%	56.9%	2.2%
other	*694*	*1.0%*	*17.7%*	*0.4%*

We observe from the fourth column that the proportion of repositories with GHA workflows varies from one language to another. It ranges from 33.2% (for Shell) to 58.5% (for TypeScript). For more recent languages such as TypeScript, Go, Kotlin, and Ruby, the majority of repositories are using GHA workflows. The top three languages (JavaScript, Python and TypeScript) together account for nearly half of all repositories in the dataset (46.6%, third column) and nearly half of the repositories defining a workflow (47.4%, last column).

Since workflows help developers to automate some of the repetitive tasks that are inherently part of the software development process, we expect larger repositories (e.g., those

[1] See https://github.com/pandas-dev/pandas/blob/68f763e7/.github/workflows/code-checks.yml for a more elaborate example of a workflow file.
[2] https://github.com/marketplace?type=actions
[3] https://docs.github.com/en/actions

TABLE II
COMPARISON OF CHARACTERISTICS FOR GITHUB REPOSITORIES WITH AND WITHOUT GHA WORKFLOWS.

	median		effect size	
characteristic	**with**	**without**	**Cliff's δ**	**interpretation**
pull requests	124	41	0.384	*medium*
contributors	20	11	0.277	*small*
commits	598	344	0.229	*small*
issues	105	59	0.227	*small*
branches	5	4	0.139	*negligible*
age (months)	71	77	−0.082	*negligible*
stars	398	334	0.078	*negligible*
size (MB)	5,878	5,099	0.025	*negligible*
forks	84	80	0.018	*negligible*
watchers	24	25	−0.013	*negligible*

having more contributors or pull requests) to rely more frequently on workflows. We compared the distributions of several characteristics between GitHub repositories with and without workflows: size in MB, number of pull requests, commits, issues, contributors, branches, stars, forks and watchers. We also considered the age of the repositories in months. A statistical difference was consistently confirmed for all characteristics ($p < 0.004$) by Mann-Whitney-U tests [34] after controlling for family-wise error rate with the Bonferroni-Holm method [35].

Table II reports the median value for each considered characteristic, as well as the effect size of the observed difference using Cliff's δ [36] and its interpretation [37]. We observe with *medium* or *small* effect size that repositories with workflows exhibit a higher number of pull requests, contributors, commits, and issues. We also observe, though with *negligible* effect size, that repositories with workflows tend to have more branches, stars, forks, and a larger size. Repositories with workflows are also younger and have fewer watchers.

More than 4 out of 10 GitHub repositories use workflows. Projects mostly written in JavaScript, Python, or TypeScript account for nearly half of the repositories and one third of the repositories using workflows. Repositories with GHA workflows tend to have more contributors, pull requests, commits, and issues.

V. WHICH KINDS OF WORKFLOWS ARE AUTOMATED?

We found a total of 70,278 workflows in 29,778 repositories, hence an average of 2.4 workflows per repository. Nearly half of these repositories (49.3%) defined a single workflow. The remaining 50.7% repositories define two (22.6%), three (11.7%), or four workflows (6.3%), even if we found dozens of repositories with 20 or more workflows.

A workflow can be triggered by one or more events. The events that trigger a workflow can be chosen from a large list of events corresponding to the different ongoing activities within a repository (e.g., commits pushed, pull requests created or updated, comments created). GHA also proposes special events such as `schedule` to execute workflows on a regular basis or `workflow_dispatch` and `repository_dispatch` to manually trigger a workflow.[4]

Table III reports the 10 most frequent events occurring in workflows and the proportion of workflows using them. Since repositories can have more than one workflow, we also report on the proportion of repositories having one of their workflows triggered by these events.

TABLE III
TOP 10 EVENT TYPES IN GITHUB WORKFLOWS AND REPOSITORIES.

event	**% workflows**	**% repositories**
`push`	41.8%	63.4%
`pull_request`	34.1%	56.3%
`workflow_dispatch`	8.3%	15.4%
`schedule`	8.1%	16.1%
`release`	3.0%	6.2%
`pull_request_target`	1.3%	2.6%
`issues`	1.0%	2.0%
`repository_dispatch`	0.7%	2.0%
`issue_comment`	0.6%	1.2%
`workflow_run`	0.4%	0.8%

The most frequent events, `push` and `pull_request`, are used by more than half of the repositories. These events correspond to what is typically monitored by traditional CI/CD services and are tightly related to the technical aspects of software development (e.g., test, build and deploy code). The next most frequent events are `workflow_dispatch` and `schedule`. The former allows to manually trigger a workflow using the GitHub API or user interface, while the latter allows triggering a workflow at a scheduled time.

The most frequent events that can be associated with more social activities are `issues` and `issue_comments`. These events, used by 2.0% and 1.2% of the repositories respectively, react to the creation of an issue or a comment. They can be used, for example, to welcome newcomers, triage issues or ensuring adherence to the *Contributor Licence Agreement* (CLA).

In order to gain some insight into the purpose of these workflows, we analysed the most common workflow names. Although the `name:` key of a workflow is optional, nearly all workflows in our dataset define it (99.1%). We found 23,056 distinct names out of the 69,618 workflows defining one. There are 554 names used by at least 10 repositories, and some of them are frequently used by a large number of repositories, such as *ci* (8.2% of the repositories), *test(s)* (5.1%), *build* (3.9%), *codeql* (3.5%), *release* (2.9%), and *lint* (1.0%). The first name explicitly referring to a “social” purpose is *mark stale issues and pull requests* and is used by 0.5% of the repositories.

Half of the repositories using GHA define two or more workflows. Workflows are mostly triggered by push and pull request events. The large majority of workflows appear to be used for technical development-related purposes.

[4]The complete list of supported events can be found on https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows.

VI. What Are the Most Frequent Jobs in Workflows?

A workflow defines one or more jobs that will be executed in parallel (by default). We found 108,500 jobs in the 70,278 workflows in the dataset, hence an average of 1.5 jobs per workflow. The large majority of workflows define a single job (77.8%) or two jobs (10.5%). However, since a repository can define multiple workflows, the number of jobs per repository is higher (3.6 on average). For instance, 65.5% of the repositories have two or more jobs defined in the totality of their workflows, and 21.3% even have 5 or more jobs.

As mentioned in Section III, a job either defines a list of steps that will be executed to achieve its purpose, or can use (through the `uses:` key) another workflow from any public repository. In the latter case, the jobs defined in the other workflow will be executed. The ability to reuse other workflows in a job is not commonly exploited by the repositories in our dataset: only 823 jobs (0.8%) from 254 distinct repositories (0.9%) refer to another workflow. Analysing the origin of the reused workflows, we found that the large majority (84.4%) have the same owner as the calling job. More specifically, 279 jobs (33.9%) use a workflow within the same repository and 416 jobs (50.5%) use a workflow located in another repository of the same owner. Only 128 jobs (15.5%) use a workflow located in another public repository.

Similar to Section V, we analysed the names of the jobs to gain insight into their purpose. However, only 40.7% of the jobs define the optional job name field. For those that do not, we considered their identifier instead (i.e., the key used to define the job). We found 32,265 distinct names, of which 851 are used by 10 or more jobs. The most frequent names ($> 1\%$) were *build* (15.5%), *test* (4.2%), *analyse* (2.2%), *lint* (2%), *release* (1.7%), and *deploy* (1.5%). The first most frequent name for a social activity was *stale* (0.9%), corresponding to the task of closing issues or pull requests that have not exhibited any recent activity.

The large majority of workflows define a single job, but most repositories have two or more jobs defined in the totality of their workflows. Jobs mostly implement technical activities. It is not common practice to reuse workflows in jobs.

VII. What Are the Automation Practices?

Steps represent the smallest unit of work in a workflow. They correspond to individual tasks in a job that are sequentially executed to achieve the job goal. For example, in order to publish a new release on a package registry such as PyPI, a job will define steps to (1) checkout the code, (2) setup Python, (3) install dependencies, (4) execute tests, (5) build the package, and (6) upload it on PyPI. We found 567,352 steps for 108,500 jobs. On average, there are 5.2 steps per job (median is 5), 8.2 steps per workflow (median is 5), and 19.4 steps per repository (median is 10).

A step either explicitly lists the commands to be executed (through the `run:` key) or delegates this task (through the `uses:` key) to an action or a Docker image. Table IV reports on the proportion of steps and repositories in function of their step type. For steps relying on the `uses:` key, we distinguish between steps referring to an action by means of a local path, a reference to a Docker image, or by specifying the name of a repository containing the action. For the latter category, we distinguish between references to the same repository, to a repository of the same owner, or to another public repository.

TABLE IV
Proportion of steps and repositories w.r.t. step type and action target.

step type	action target	% steps	% repositories
`run:`		49.9%	93.5%
`uses:`	local path	0.8%	2.0%
	Docker image	0.1%	1.8%
	same repository	0.2%	0.4%
	same owner	0.7%	4.3%
	other repository	48.3%	99.3%

Around half of the steps (51.1%) use an *action*. Nearly all repositories have at least one step referring to an action. We observe that these steps mostly refer to another public repository (48.3% of the steps and 99.3% of the repositories) and, to a much lower extent, to a repository belonging to the same owner (0.7% of the steps and 4.3% of the repositories). There is little to no use of Docker images (0.1% of the steps).

The other half of the steps (49.9%) run their own local commands. Analysing the content of `run:` keys, we found 139,501 distinct commands among the 287,868 steps that define one. There are 134 commands being duplicated in 100+ steps, of which 10 are duplicated in more than 1,000 steps (e.g., `npm ci`, `yarn build`, `npm test`). On average, steps execute 2.9 command lines (median is 1) for a total of 29.8 command lines per repository (median is 4).

Looking at the step names, we found 92,853 distinct names of which 360 are used by at least 100 steps. The most frequent ($> 1\%$) names are *install dependencies* (3.7%), *checkout* (2.9%), *build* (2%), *run tests* (1.4%), *test* (1.3%), *checkout repository* (1.2%), and *checkout code* (1.1%). When we distinguish names based on the type of the steps using them, we find that the use of some step names is strongly related to the presence of commands (i.e., `runs:`) or to the reuse of an existing action (i.e., `uses:`). For example, we found that *install dependencies*, *build*, and *tests* (including their variants) are mostly used for steps executing commands ($\geq 97\%$ of the steps with these names) while *checkout* (and its variants) and *setup python|node|php|etc.* are mostly used for steps relying on a reusable action ($\geq 99\%$).

Overall, considering the names being used in at least 100 steps, we found that 47.2% of them are specialized (i.e., $\geq 75\%$ steps having that name) towards steps executing commands, and 45.3% towards steps using an action. The remaining 7.5% names are used indiscriminately for commands and actions, and include terms like *deploy*, *release*, and *publish* among others.

A job defines five steps on average. Half of the steps are running their own local commands while the other half use a reusable *action*. Nearly all repositories rely on actions, mostly originating from other public repositories. A large majority of the most frequent tasks are implemented either almost exclusively by steps executing commands or almost exclusively by an action.

VIII. Which Actions Are Reused and How?

Section VII revealed the common practice of steps using reusable actions. From the 278,122 steps relying on a reusable action from a public repository, we found 2,964 distinct actions of which 724 are used at least 10 times. Table V lists the 10 most frequent actions observed in steps and repositories.

TABLE V
The 10 most frequent actions in steps and repositories.

action	% steps	% repositories
actions/checkout	35.5%	97.8%
actions/cache	7.2%	21.6%
actions/setup-node	6.6%	26.3%
actions/upload-artifact	5.9%	18.7%
actions/setup-python	5.8%	21.0%
actions/setup-java	2.6%	10.0%
actions/setup-go	2.5%	9.1%
actions/download-artifact	2.1%	6.4%
shivammathur/setup-php	1.3%	5.7%
codecov/codecov-action	1.3%	9.4%

The most frequently used action is actions/checkout, used by 35.5% of the steps and 97.8% of the repositories. The high proportion of repositories using it should not be surprising since actions/checkout aims to ease checking out a repository, a necessary first step for executing most of the CI/CD tasks. Other frequently used actions are mainly related to the deployment of a specific programming language environment (e.g., setup-node or setup-python). Overall, 24.2% of the steps use an action of the form setup-*.

For each action in the dataset, we identified its provider (i.e., the name of the user or organization owning the repository). Considering the top 10 of Table V, we observe that 8 of the most frequent actions are officially proposed by GitHub (distributed by the actions provider) and used by 71.2% of the steps. The first third-party provider is shivammathur with the setup-php action.

In the entire dataset we found 2,037 different providers, but the majority of the actions used by steps come from just a few providers. There are only 8 providers whose actions are called by 1,000+ steps, and 103 providers whose actions are called by 100+ steps. The actions provider alone distributes 24 actions that account for 71.7% (i.e., 199,549) of the steps calling an action. The second most frequent called provider is docker (7 actions and 3.8% steps), followed by github (9 actions and 3.1% steps) mostly for their CodeQL actions, and by shivammathur (2 actions and 1.4% steps) for its setup-php action.

We sought to find out more about the purpose of these actions. However, the metadata that are required to distribute an action on a public repository do not include anything about the action's purpose or category.[5] The only place where this information is (partially) available is on the GitHub Marketplace.[6] To publish an action on the Marketplace, one has to provide additional metadata, including the primary and secondary categories the action belongs to. Even if GitHub allows anyone to upload to the Marketplace an action available from a public repository, so as to increase its visibility and reuse, not all actions are published on the Marketplace. We managed to find 917 of the 2,964 actions of our dataset on the Marketplace, based on the assumption that the name of the repository where the action is developed corresponds to its unique identifier on the Marketplace. We will discuss the threats related to this assumption in Section XI. Out of these 917 actions we found, 752 were correctly mapped, in the sense that the repository mentioned on the Marketplace for an action does indeed correspond to the repository that is called by the step. These 752 actions are used in 158,441 steps (i.e., 57% of the steps relying on a public action).

Table VI reports on the most frequent ($> 1\%$) categories for these actions, as well as on the proportion of steps and repositories using them. Since we did not observe major differences between primary and secondary categories, we report exclusively on the former.

TABLE VI
Most frequent ($> 1\%$) primary categories and the proportion of actions, steps and repositories using them.

action category	% actions	% steps	% repo.
Utilities	23.9%	88.2%	99.4%
Continuous integration	17.3%	4.9%	12.9%
Publishing	7.2%	0.7%	2.9%
Deployment	6.9%	0.2%	0.6%
Code quality	6.1%	0.3%	1.0%
Project management	5.2%	0.4%	1.6%
Dependency management	4.4%	1.0%	2.5%
Code review	4.1%	0.1%	0.6%
Testing	3.3%	0.7%	1.5%
Open Source management	3.3%	0.1%	0.5%
Container CI	2.3%	2.2%	5.5%
Chat	1.9%	0.3%	0.5%
Reporting	1.7%	0.1%	0.4%
Community	1.6%	0.0%	0.1%
Security	1.6%	0.1%	0.3%
unspecified	4.9%	0.2%	0.6%

Table VI reveals that most categories correspond to technical aspects of software development. The primary categories containing the highest proportion of actions (as well as steps and repositories) are *Utilities* (23.9% of actions) and *Continuous integration* (13.3% of actions). These two "catch-all" categories include very diverse actions to check out repositories, set up environments, create releases, etc. The third most widely used category is *Container CI* even if only 2.3% of the actions are part of this category. It includes actions to log in to a Docker registry, to run a Dockerfile, or to set up specific services (e.g., a MySQL database or a Redis instance). A

[5]https://docs.github.com/en/actions/creating-actions/metadata-syntax-for-github-actions
[6]https://github.com/marketplace?type=actions

few categories include socially related actions, such as *Project management* and *Code review*. These two categories notably propose actions to create or triage issues, to detect and lock stale issues, or to add specific comments in existing issues or pull requests. Other social categories include *Chat*, *Reporting* and *Community*. The two former ones propose actions to notify on Slack, Discord, IRC, etc. while the latter includes actions to ensure CLA adherence or to welcome newcomers.

Using actions in steps is a common practice. A few actions concentrate most of the reuse, and most of them are distributed by GitHub and belong to the *Utilities* or *Continuous integration* categories.

IX. Which versioning practices are being used?

When a step uses a predefined action, in addition to specifying the name of the repository hosting the action it can optionally specify which *version* of the action should be executed, by means of a git reference. This reference can be a commit SHA (e.g., `@753c60e0`), a branch name (e.g., `@main`), or a git tag (e.g., `@v2.1.3`). If no reference is specified, the latest version of the action is executed. For security and stability reasons, GitHub recommends pinning an action to a full-length commit SHA. While this is the most secure option, specifying a tag is more convenient since GitHub's release management support advocates the creation of a tag corresponding to the version number of a new release. This makes it easy for a step using an action to specify which version of the action should be executed.

Table VII shows the number and proportion of steps in function of the git reference type used for actions, and the origin of these actions. We distinguish between no git reference (*none*), a reference to a specific *commit SHA*, a reference to a *version tag* and a reference to a branch or another tag (*branch/tag*).

TABLE VII
Number and proportion of steps w.r.t. action target and git reference type.

		steps		
action origin	**reference type**	**#**	**% target**	**% all**
local path	*none*	4,397	100.0%	1.5%
	branch/tag	2	0.0%	0.0%
same repository	commit SHA	2	0.2%	0.0%
	version tag	123	12.8%	0.0%
	branch/tag	833	87.0%	0.3%
same owner	*none*	1	0.0%	0.0%
	commit SHA	123	3.0%	0.0%
	version tag	2,641	63.9%	0.9%
	branch/tag	1,368	33.1%	0.5%
other repository	*none*	2	0.0%	0.0%
	commit SHA	4,601	1.7%	1.6%
	version tag	258,647	93.0%	89.9%
	branch/tag	14,872	5.3%	5.2%

We observe major differences in the git references used to refer to an action in function of the origin of the action. For instance, actions on a local path are nearly exclusively referred to without any specific reference, while the vast majority of actions within the same repository (but referred to with the full name of the repository) are referred to with a branch or a tag name or, to a much lower extent, with a version tag. The opposite can be observed for actions from repositories of the same owner: they are mostly referred to using a version tag and, to a lower extent, with a branch or a tag name. The situation is even more marked for actions coming from other public repositories: 9 out of 10 steps in this large subset use a version tag to refer to an action.

Assuming adherence to *semantic versioning*, GitHub recommends specifying the version tag by including only its major component (e.g., `@v2` instead of `@v2.1.3`) in order to receive critical fixes and security patches while still maintaining compatibility. This recommendation seems to be widely followed, since 89.9% of the version tags used to refer to an action include only a major component (e.g., `@v2`), 0.9% a minor component (e.g., `@v2.1`) and 9.2% a patch component (e.g., `@v2.1.3`).

Referring to a version using only its major component has clear advantages if we assume adherence to semantic versioning [10]. However, since versions of an action are identified using git tags, this means that the action maintainer must *move* some of these tags each time a new version of the action is released (e.g., moving `@v2` and `@v2.1` from `@v2.1.3` to `@v2.1.4` when version `2.1.4` is released). Unless automated, this introduces an additional burden on the maintainers. Forgetting to update these tags when a minor or a patch update is released for an action implies that the workflows that depend on it do not automatically benefit from the bug and security fixes provided by the update.

Moreover, we found that 16.4% of the major components used in version tags to refer to a reusable action do not target the highest major release of the corresponding action, i.e., they are relying on a lower major train. As such, dependent workflows do not benefit from the latest bug and security fixes of the action unless these changes are backported to lower major trains as well [11].

Around 9 out of 10 steps use a version tag when referring to an action. Most of the version tags only specify the major component of the targeted version, and one sixth of them refer to a lower major train.

X. Discussion

On the Popularity of GHA

The growing popularity of GHA is undeniable. The findings from Section IV revealed that a significant proportion of repositories rely on GHA, regardless of their main programming language. These findings are confirmed by Golzadeh et al. [8] who showed that in 18 months, GHA has become the dominant CI/CD service on GitHub. They attributed this popularity to a combination of factors, including the deep integration of GHA with GitHub, the ease of use, the speed, its free tier for open source projects, the availability of a large marketplace of reusable actions, and the support for many different operating systems.

We contacted five developers using GHA, who confirmed that these aspects played a major role in their decision to adopt GHA: "*we migrated to GHA because it was directly integrated with GitHub and had all the same support across Linux, Mac, Windows that we were looking for*", "*the integration of GHA into GitHub itself would be the reason of its popularity*", "*you have this marketplace of GHA and you can reuse components from there, and it's pretty straightforward and easy*", "*People are using GHA for the simplicity, the speed, because it's free, because you have access to macOS and Windows runners [...]*" and "*I'm just looking for something that I can set up with the absolute least friction, and in 99 times out of 100 that's going to be GitHub Actions.*"

Travis, the previous dominant CI/CD tool on GitHub, has been encountering many issues in recent years, such as restrictions on their free plan, decreased reliability, poor Docker support, long build times and lack of innovation [29]. This has further contributed to the success of GHA, as shown by Golzadeh et al. [8] who observed that most migrations between CI services go from Travis towards GHA. This is confirmed by developers that we have talked to: "*Travis was really a game changer 10 years ago, [but] did not really evolve [or] bring any innovation, [...] so we moved away from Travis*" and "*since Travis has been bought by a bigger company, it pretty much became useless and we moved away from Travis in all our projects to GHA*".

Another important milestone for GHA relates to GitHub being acquired by Microsoft in 2018. A developer pointed out that "*because Microsoft is now the mother company of GitHub, you see a lot of investment taking place in GitHub. The features being added are actually coming from a lot of areas from within Microsoft*". Another developer signals that GHA is one of these features: "*Just look at the free GitHub Actions nowadays. That's a lot of money they are spending for free so everybody can build stuff.*" Most developers we talked to are positively surprised by Microsoft's changed attitude regarding open source, for example: "*These last years, Microsoft is really doing huge changes internally to make their reputation change about open source. I think Microsoft is changing its point of view on open source and I think it's for the greater good of open source developers.*" Despite these positive signs, having a private company owning the dominant platform for distributing open source software remains risky since it implies a *de facto* monopoly. At some point in the future, Microsoft might change its strategy to try to make profit out of the situation: "*There are intangible benefits that Microsoft gets and we'll see if changes happen in the next few years to where GitHub makes changes to be more profitable that don't necessarily serve the free software folks.*" and "*I have mixed feelings about it, on the one hand, it really is convenient having everything integrated at one place. On the other hand, how much do we really want to invest all of open source in a single company?*" This may pose difficulties to many OSS projects hosted on GitHub, as they will be facing some kind of vendor lock-in. Although GHA supports self-hosted runners, many aspects of GHA (including the workflow syntax or the reusable actions) are tightly coupled to GitHub and migrating away from GHA could take a lot of effort. Only time will tell how this situation will affect the future of OSS development.

The GHA Ecosystem

The findings reported in Section VIII have revealed that the ability to reuse actions in GHA has lead to a new emerging ecosystem that is worthy of being studied in its own right. The reuse of actions in steps is a common practice that allows developers to easily integrate (sometimes complex) tasks without having to code them. The ability for a CI/CD tool to provide reusable components is not something specific to GHA, since many competing CI/CD tools are providing similar mechanisms. For instance, CircleCI introduced *orbs* in 2018, one year before GHA[7]; and Jenkins has been providing community-contributed plugins for years through plugins.jenkins.io. However, at the time of writing, GHA offers more than 12,000 reusable components on its Marketplace, about 4 times as many as CircleCI orbs and more than 6 times higher than Jenkins plugins, and there are likely thousands more available actions in public GitHub repositories.

Given this wide availability of reusable actions, GHA should be studied as a "software ecosystem" that bears many similarities to ecosystems of reusable software libraries distributed by package managers, e.g., npm, Cargo, RubyGems, Maven, PyPI and the like. The parallel with such packaging ecosystems is quite obvious: automated workflows, as software *clients*, frequently express *dependencies* towards reusable actions that can exist in different *versions* or *releases*. Packaging ecosystems are known to suffer from a large number of issues in the reusable artefacts they distribute, and each of these has been an active topic of investigation. Well-known challenges include obsolescence or outdatedness [38], [39], dependency issues [9], [40], [41], breaking changes [10], [14], security vulnerabilities [2], [12], and so on.

The GHA ecosystem is likely to suffer from very similar issues, and these issues will continue to become more important and more impactful, as the number of reusable actions continues to grow at a rapid pace. Therefore, there is an urgent need for further research as well as appropriate tooling to support developers of reusable actions and workflows, especially since these issues may not only affect the workflows but also wide ranges of projects that use them. A first step in this direction is GitHub's built-in Dependabot dependency monitoring service that has recently (January 2022) started to support actions.

Security Concerns

While security concerns are important to deal with for any software project, it is known that the attack surface of security issues has become several orders of magnitude higher due to the widespread dependence on reusable software libraries that can have deep transitive dependency chains [2], [12], [42], [43]. The reliance on CI/CD tools to automate development

[7]https://circleci.com/blog/announcing-orbs-technology-partner-program/

activities in software projects continues to increase this attack surface considerably. Two developers that we talked to confirm that this constitutes a major security concern: *"In house we have two GitHub and GitLab CI systems that we need to maintain. It's a major security concern because, from that automation you can basically run any code on it."* and *"CI/CD are very important to secure the build chain, that we should focus in the future into the aspect of securing the toolchain."*

For GHA in particular, multiple examples of security issues with potentially disastrous consequences have been reported, such as manipulating pull requests to steal arbitrary secrets,[8] injecting arbitrary code with workflow commands[9] or bypassing code reviews to push unreviewed code.[10] A developer we talked to specifically mentioned "*You can open a pull request, build the package, and then we will deliver it. And when you do that from a pull request, there are issues with the security considerations about the credentials, because anyone could modify workflows or inject code, get access to the credentials and then access to the upload process. [...] When it's open to everyone, you need to be careful. [...] What we do with GitHub Actions, we build in one workflow, and we have a second workflow which does the upload or the shipping or the delivery with credentials which are not exposed in the build pipeline for the pull request.*"

Relying on reusable actions from third-party repositories or even from the Marketplace further increases the vulnerability attack surface. Since a job executes its commands within a runner shared with other jobs from the same workflow, individual jobs in a workflow can compromise other jobs they interact with. For example, a job could query the environment variables used by a later job, write files to a shared directory that a later job processes, or even more directly interact with the Docker socket and inspect other running containers and execute commands in them.[11]

Despite these risks, Section VIII revealed that it is common practice to rely on reusable actions. As a general rule of thumb, GitHub recommends to only use actions whose creator can be trusted. However, even actions from trusted creators can be compromised. For example, an attacker having gained write access to the repository of a trusted action can change its code and commands in order to compromise the repositories depending on this action.

To further reduce the risks of using compromised actions, GitHub suggests referring to reusable actions through their unique commit SHA, to avoid unintentionally using a comprised action that may have its code changed and may be able to steal secrets: *"Pinning to a particular SHA helps mitigate the risk of a bad actor adding a backdoor to the action's repository, as they would need to generate a SHA-1 collision for a valid Git object payload."*[12] Unfortunately, we observed in Section IX that this recommendation is not really followed in practice. The very large majority of steps relying on a reusable action use a version tag (rather than a commit SHA) when referring to an action, and most of the version tags only specify the major component of the targeted version, implying that any new compromised version of an action will be automatically adopted by most dependent workflows.

We are not aware of any publicly available quantitative analysis having reported on the impact of reusable actions on security vulnerabilities in software projects. Such empirical research is urgently needed, in order to quantify the extent of the problem. That would constitute a first step towards trying to reduce the attack surface of vulnerabilities related to the use of GHA.

XI. Threats to Validity

We follow the structure recommended by Wohlin et al. [44] to discuss the main threats to validity of our research.

Threats to *construct validity* concern the relation between the theory behind the experiment and the observed findings. They can be mainly due to imprecisions in the measurements we performed. We detected the use of automated workflows in GitHub repositories on the basis of the presence of a YAML file in the `.github/workflows` folder. This approach leads to an overestimation since the presence of a YAML file does not necessarily imply that the corresponding workflow is actually being triggered and used. However, we are confident that such workflows are indeed used in the vast majority of cases as there is little to no practical reason to keep workflows in `.github/workflows` without using them.

Another threat to construct validity stems from how we identified the git reference type used to reference public actions in steps. We relied on a heuristic to detect whether the git references correspond to a version number (via a regular expression), to a commit SHA (based on the git reference length) or to a tag or branch name (all remaining git references). This naive heuristic seemed to be effective since we did not find any false positives after having manually checked 104 distinct randomly selected cases.

Another threat to validity stems from how we interpreted the identifier labels for workflows and jobs. We relied on these labels in Sections V and VI to understand the purpose of the workflows and jobs being defined. While some workflows and jobs may have a label that does not reflect their purpose, we believe these cases to be rare, as the goal of a label is to provide an indication of the nature of the workflow or job, and not to mislead practitioners or researchers.

Threats to *internal validity* concern choices and factors internal to the study that could influence the observations we made. One of such choices relates to how we mapped actions used in steps with the ones distributed on the GitHub Marketplace. We relied on the assumption that the name of

[8]https://blog.teddykatz.com/2021/03/17/github-actions-write-access.html

[9]https://packetstormsecurity.com/files/159794/GitHub-Widespread-Injection.html

[10]https://medium.com/cider-sec/bypassing-required-reviews-6e1b29135cc7

[11]https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions#using-third-party-actions

[12]https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions

the repository where an action is developed corresponds to its unique identifier on the Marketplace. This led both to false positives (e.g., action myci-actions/checkout is identified as checkout while checkout on the Marketplace is provided by actions/checkout) and false negatives (e.g., action actions/setup-node is identified as setup-node-js-environment on the Marketplace). False positives were addressed by comparing the repository referred from the calling step with the repository listed on the action page on the Marketplace. We were unable to address the false negatives, as this would require extracting all actions from the Marketplace in order to obtain their development repositories. Unfortunately, GitHub does not provide a complete list of actions in the Marketplace, nor an API to obtain them. We remain confident that the findings of Section VIII are representative, since we managed to map correctly 727 actions, accounting for 57% of the steps relying on an action.

Threats to *conclusion validity* concern the degree to which the conclusions derived from our analysis are reasonable. Since our conclusions are mostly based on quantitative observations, they are unlikely to be affected by such threats.

Threats to *external validity* concern whether the results can be generalized outside the scope of this study. One such threat was our decision to study active repositories having at least 100 stars and 100 commits, aiming at excluding abandoned, personal or experimental repositories that do not necessarily correspond to software development [32]. This implies that we have no insight into the use of GHA in smaller or less active repositories, and it could be the case that GHA is used for different purposes in those repositories (e.g., to publish GitHub pages or to compile LaTeX files).

XII. Conclusion

GHA has become the dominant CI/CD on GitHub, only 18 months after its introduction. In order to get a deeper insight into the GHA ecosystem, we have conducted a quantitative study of 29,778 GitHub repositories containing 70,278 GHA workflows. We characterised these repositories and their workflows, in terms of which jobs, steps and reusable actions were used and how. We observed that workflows tend to be used in the more active GitHub projects (more contributors, pull requests, commits and issues). These workflows and their jobs are primarily used for development purposes, and mostly triggered by push or pull request events. About half of all steps in jobs rely on reusable actions, mostly originating from public repositories. Most of the actions being reused are provided by GitHub itself, and their primary purpose is for continuous integration and other utilities. Actions appear to be reused by adhering to some sort of semantic versioning. The reuse of actions can be problematic, as it has the potential of increasing the attack surface of security issues by several orders of magnitude.

This article is the first to have provided a large-scale quantitative assessment of GitHub repository workflows relying on GitHub Actions. We have quantified the widespread use of GHA workflows and reusable actions in GitHub repositories. Nevertheless, we only scratched the surface of the emerging GHA ecosystem. More in-depth empirical studies remain required to provide a comprehensive understanding of the GHA ecosystem. Such studies should focus on assessing, in the context of the GHA ecosystem, the prevalence and impact of the well-known challenges and issues encountered by more traditional software ecosystems. Indeed, in this new context, those issues may not only affect the GHA workflows but also a wide range of projects using them. Additional work is also required to study the security concerns related to the reuse of actions, as well as how this ecosystem is evolving over time, and how it compares to similar and related CI/CD ecosystems (such as the ecosystem of orbs on CircleCI, of plugins on Jenkins, or the Infrastructure as Code ecosystem of roles in RedHat Ansible [45]).

As future work, we plan to conduct a complementary qualitative analysis by interviewing experienced practitioners to identify and understand the main motivations for adopting and using GHA and the perceived benefits and drawbacks of doing so. Such qualitative insights aim to complement the quantitative results of this paper in order to further increase the understanding of the emerging GHA ecosystem and its implications on collaborative OSS development in GitHub.

Specifically related to the research question of which actions are reused and how, we plan to study the versioning practices followed by actions and by the workflows relying on them, considering how frequently actions and workflows are updated, and to which extent workflows stay up-to-date with respect to the actions they are using.

Acknowledgment

This research is supported by the ARC-21/25 UMONS3 Action de Recherche Concertée financée par le Ministère de la Communauté française – Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique, as well as by the Fonds de la Recherche Scientifique - FNRS under grant numbers O.0157.18F-RG43, T.0017.18 and T.0022.13.

References

[1] J. M. Costa, M. Cataldo, and C. R. de Souza, "The scale and evolution of coordination needs in large-scale distributed projects: implications for the future generation of collaborative tools," in *SIGCHI Conference on Human Factors in Computing Systems*, 2011, pp. 3151–3160.

[2] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *15th international conference on mining software repositories*, 2018, pp. 181–191.

[3] A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency issues in OSS packaging ecosystems," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017.

[4] G. Avelino, E. Constantinou, M. T. Valente, and A. Serebrenik, "On the abandonment and survival of open source projects: An empirical investigation," in *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–12.

[5] GitHub, "The 2021 state of the octoverse - community report," 2021. [Online]. Available: octoverse.github.com

[6] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, “Social coding in GitHub: Transparency and collaboration in an open software repository,” in *International Conference on Computer Supported Cooperative Work (CSCW)*. ACM, 2012, pp. 1277–1286.

[7] C. Chandrasekara and P. Herath, *Hands-on GitHub Actions: Implement CI/CD with GitHub Action Workflows for Your Applications*. Apress, 2021.

[8] M. Golzadeh, A. Decan, and T. Mens, “On the rise and fall of CI services in GitHub,” in *29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021.

[9] A. Decan, T. Mens, and P. Grosjean, “An empirical comparison of dependency network evolution in seven software packaging ecosystems,” *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.

[10] A. Decan and T. Mens, “What do package dependencies tell us about semantic versioning?” *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1226–1240, 2019.

[11] A. Decan, T. Mens, A. Zerouali, and C. De Roover, “Back to the past–analysing backporting practices in package dependency networks,” *IEEE Transactions on Software Engineering*, 2021.

[12] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, “Small world with high risks: A study of security threats in the npm ecosystem,” in *28th USENIX Security Symposium*, 2019, pp. 995–1010.

[13] M. Valiev, B. Vasilescu, and J. Herbsleb, “Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem,” in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018, pp. 644–655.

[14] J. Dietrich, D. Pearce, J. Stringer, A. Tahir, and K. Blincoe, “Dependency versioning in the wild,” in *16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 349–359.

[15] M. Fowler and M. Foemmel, “Continuous Integration,” https://martinfowler.com/articles/originalContinuousIntegration.html, September 2000, [Online; accessed 3 January 2022].

[16] 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*, pp. 1–1, 2021.

[17] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, “Usage, costs, and benefits of continuous integration in open-source projects,” in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 426–437.

[18] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, “Quality and productivity outcomes relating to continuous integration in GitHub,” in *Joint Meeting on Foundations of Software Engineering (FSE)*, 2015, pp. 805–816.

[19] H. Lamba, A. Trockman, D. Armanios, C. Kästner, H. Miller, and B. Vasilescu, “Heard it through the Gitvine: An empirical study of tool diffusion across the npm ecosystem,” in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2020, pp. 505–517.

[20] E. Soares, G. Sizilio, J. Santos, D. Alencar, and U. Kulesza, “The effects of continuous integration on software development: a systematic literature review,” *Empirical Software Engineering*, 2022.

[21] B. Vasilescu, S. van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. van den Brand, “Continuous integration in a social-coding world: Empirical evidence from GitHub,” in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 401–405.

[22] F. Zampetti, S. Geremia, G. Bavota, and M. Di Penta, “CI/CD pipelines evolution and restructuring: A qualitative and quantitative study,” in *International Conference on Software Maintenance and Evolution (ICSME)*, 2021.

[23] N. Cassee, B. Vasilescu, and A. Serebrenik, “The silent helper: The impact of continuous integration on code reviews,” in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 423–434.

[24] 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.

[25] T. Durieux, R. Abreu, M. Monperrus, T. F. Bissyandé, and L. Cruz, “An analysis of 35+ million jobs of Travis CI,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 291–295.

[26] M. Beller, G. Gousios, and A. Zaidman, “Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub,” in *International Conference on Mining Software Repositories (MSR)*, 2017, pp. 356–367.

[27] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, “The impact of continuous integration on other software development practices: A large-scale empirical study,” in *International Conference on Automated Software Engineering (ASE)*, 2017, pp. 60–71.

[28] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu, “Wait for it: Determinants of pull request evaluation latency on GitHub,” in *Working Conference on Mining Software Repositories (MSR)*, 2015, pp. 367–371.

[29] 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 (ESEC/FSE)*. ACM, 2019, pp. 647–658.

[30] T. Kinsman, M. Wessel, M. A. Gerosa, and C. Treude, “How do software developers use GitHub Actions to automate their workflows?” in *International Conference on Mining Software Repositories (MSR)*, 2021.

[31] P. Valenzuela-Toledo and A. Bergel, “Evolution of GitHub Action workflows,” in *29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022.

[32] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, “The promises and perils of mining GitHub,” in *International Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 92–101.

[33] O. Dabic, E. Aghajani, and G. Bavota, “Sampling projects in GitHub for MSR studies,” in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 2021, pp. 560–564.

[34] H. B. Mann and D. R. Whitney, “On a test of whether one of two random variables is stochastically larger than the other,” *Ann. Math. Statist.*, vol. 18, no. 1, pp. 50–60, 03 1947.

[35] S. Holm, “A simple sequentially rejective multiple test procedure,” *Scandinavian Journal of Statistics*, vol. 6, no. 2, pp. 65–70, 1979. [Online]. Available: http://www.jstor.org/stable/4615733

[36] N. Cliff, “Dominance statistics: Ordinal analyses to answer ordinal questions,” *Psychological bulletin*, vol. 114, no. 3, p. 494, 1993.

[37] 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.

[38] A. Decan, T. Mens, and E. Constantinou, “On the evolution of technical lag in the npm package dependency network,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 404–414.

[39] F. Cogo, G. Oliva, and A. E. Hassan, “Deprecation of packages and releases in software ecosystems: A case study on npm,” *IEEE Transactions on Software Engineering*, 2021.

[40] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, “Do developers update their library dependencies?” *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[41] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, “A comprehensive study of bloated dependencies in the Maven ecosystem,” *Empirical Software Engineering*, vol. 26, no. 3, p. 45, 2021. [Online]. Available: https://doi.org/10.1007/s10664-020-09914-8

[42] M. Alfadel, D. E. Costa, E. Shihab, and E. Shihab, “Empirical analysis of security vulnerabilities in Python packages,” in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021.

[43] J. Düsing and B. Hermann, “Analyzing the direct and transitive impact of vulnerabilities onto different artifact repositories,” 2021.

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

[45] R. Opdebeeck, A. Zerouali, C. Velázquez-Rodríguez, and C. De Roover, “On the practice of semantic versioning for Ansible Galaxy roles: An empirical study and a change classification model,” *Journal of Systems and Software*, vol. 182, 2021.