# On the Outdatedness of Workflows in the GitHub Actions Ecosystem

Alexandre Decan[1], Tom Mens, Hassan Onsori Delicheh

[a]*Software Engineering Lab, University of Mons, Mons, Belgium*

**Abstract**

GitHub Actions was introduced as a way to automate CI/CD workflows in GitHub, the largest social coding platform. Thanks to its deep integration into GitHub, GitHub Actions can be used to automate a wide range of social and technical activities. Among its main features, it allows automation workflows to rely on reusable components – the so-called Actions – to enable developers to focus on the tasks that should be automated rather than on how to automate them. As any other kind of reusable software components, Actions are continuously updated, causing many automation workflows to use outdated versions of these Actions. Based on a dataset of nearly one million workflows obtained from 22K+ repositories between November 2019 and September 2022, we provide quantitative empirical evidence that reusing Actions in GitHub workflows is common practice, even if this reuse tends to concentrate on a limited number of Actions. We show that Actions are frequently updated, and we quantify to which extent automation workflows are outdated with respect to these Actions. Using two complementary metrics, technical lag and opportunity lag, we found that most of the workflows are using an outdated Action release, are lagging behind the latest available release for at least 7 months, and had the opportunity to be updated during at least 9 months. This calls for a more rigorous management of Action outdatedness in automation workflows, as well as for better policies and tooling to keep workflows up-to-date.

*Keywords:* software ecosystem, dependency management, continuous integration, collaborative software development, workflow automation, technical lag

*Email addresses:* `alexandre.decan@umons.ac.be` (Alexandre Decan), `tom.mens@umons.ac.be` (Tom Mens), `hassan.onsoridelicheh@umons.ac.be` (Hassan Onsori Delicheh)

[1]F.R.S.-FNRS Research Associate

## 1. Introduction

Over 80% of today's software is either open source or is depending on it to a large extent.[2] Open source software development has become a continuous, highly distributed and collaborative endeavour [1]. A wide range of development-related tasks must be carried out during collaborative software development: developing, debugging, testing and reviewing code; quality and security analysis; packaging, releasing and deploying software distributions; and so on. This makes it increasingly challenging for contributor communities to keep up with the rapid pace of producing and maintaining high-quality software releases. It requires the orchestrated use of tools such as version control systems, software distribution managers, bug and issue trackers, vulnerability and dependency analysers. These tools tend to be integrated into social coding platforms that have revolutionised collaborative software development practices in the last decade. A well-known example is GitHub, which is by far the largest social coding platform, hosting millions of software repositories, and accommodating over 94 million users in 2022 [2].

To keep up with the rapid pace of producing and maintaining high-quality software releases, automation workflows were introduced to automate numerous repetitive activities that are part of the development process [3]. Continuous integration, deployment and delivery (CI/CD) have become the cornerstone of collaborative software development and DevOps practices. While CI/CD services (e.g., Travis or Jenkins) have been in widespread use for over a decade, they became tightly integrated into the most popular social coding platforms, effectively transforming the development workflow automation landscape [4]. GitHub publicly announced the beta version of GitHub Actions (hereafter abbreviated GHA) in October 2018 to support CI/CD for GitHub repositories, allowing developers to automate their workflows directly within GitHub. GitHub officially released GHA in November 2019.

The deep integration of GHA into GitHub enables to use this technology for a wide range of services including CI/CD, code reviewing, communication with developers, verifying licence agreements, and monitoring and fixing dependencies and security vulnerabilities. GHA workflows can be triggered by a variety of events, including commits, issues, pull requests, comments, schedules, releases and tags [5]. Workflow developers have the choice between defining sequences of commands to be executed or using Actions as reusable components. These Actions are developed and distributed in public GitHub repositories and on the GitHub Marketplace.[3]

Since its public release, GHA has become the dominant CI/CD service on GitHub, only 18 months after its introduction [4]. Its Marketplace of reusable Actions has been growing very rapidly ever since, reaching 16K Actions in December 2022. Given the wide availability and use of Actions [6], GHA bears

[2]`https://www.linuxfoundation.org/blog/chaoss-project-creates-tools-to-analyze-software-development-and-measure-open-source-community-health/`

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

many similarities to ecosystems of reusable software libraries (such as npm, RubyGems, Maven or PyPI) [7]. These library ecosystems are known to suffer from various issues related to dependency management [8, 9, 10], security vulnerabilities [11, 12, 13], backward compatibility [14, 15, 16], outdated components [17, 18, 19], deprecated and obsolete components [20] and loss of core developers [21]. Wessel et al. [22] extensively argue that GHA is forming a similar, yet quite distinct, ecosystem that is likely to suffer from very similar issues.

This article focuses on one of these issues in more detail, namely the outdatedness of workflows induced by the Actions used by them. Just like reusable software libraries, Actions are continuously updated, and workflow maintainers are confronted with the difficult choice of whether, when, and how to keep these components up to date [20].

On the one hand, maintainers would ideally adopt the most recent updates of the components they use as soon as they become available, in order to benefit from the latest bug and security fixes. On the other hand, reusable components may cause workflows to become outdated because workflow maintainers may not be aware of available updates [23]. Moreover, maintainers sometimes consciously choose not to update because they feel they do not need the new functionality ("*if it ain't broke, don't fix it*"), because upgrading exposes to an increased risk of breaking changes, or because of the cost and effort required to update. This conscious choice increases the risk of having bugs and security issues that may have known fixes [24, 25]. Infamous examples include the equifax and Log4Shell incidents, and vulnerable and outdated components are nowadays acknowledged as being a major security risk [26].

The same is likely to hold for GHA workflows relying on outdated Actions. According to GitHub Security Lab, "*a compromised or malicious Action could potentially disrupt automatic workflows of your repository*" and "*all options [to refer to an Action ] are a trade-off between guaranteed supply chain integrity and auto-patching of vulnerabilities in dependencies*".[4] Despite its recency, GHA already exhibited several examples of Action releases suffering from security issues for which fixes have been made available in newer releases.[5]

In this article, based on a dataset of nearly one million workflows obtained from 22K+ repositories between November 2019 and September 2022, we address two main research goals related to the reuse of Actions by GitHub workflows and the outdatedness resulting from such reuse. We decided to carry out an evolutionary analysis, in order to determine whether any significant changes could be observed over time.

**$G_1$** The first research goal aims to provide quantitative insights on how workflows in GitHub repositories tend to rely on reusable Actions over time. To do so, we study three research questions:

[4]`https://securitylab.github.com/research/github-actions-building-blocks/`
[5]`https://github.com/advisories?query=ecosystem%3Aactions`

$RQ_1$ *To which extent do workflows rely on reusable Actions?* This question aims to highlight the prevalence of reusable Actions in GitHub workflows. We show that relying on reusable Actions is common and that a limited number of Actions concentrate most of the reuse.

$RQ_2$ *How frequently are Actions and workflows updated?* This question focuses on how workflows and Actions evolve over time. We show that Actions and workflows are continuously updated.

$RQ_3$ *Which versioning practices are being used in GHA?* This question studies whether workflows and Actions adhere to a semantic versioning policy or whether other versioning practices are being used. We show that most Actions follow a three-component versioning notation and that GitHub's recommendation for semantic versioning is widely followed.

**$G_2$** The second research goal focuses on quantifying the outdatedness of workflows in terms of the reusable Actions used by them. We again study three research questions:

$RQ_4$ *How prevalent are outdated workflows?* This question aims to highlight the prevalence of workflows relying on outdated releases of Actions. We show that most of the workflows are using an outdated release of an Action.

$RQ_5$ *What is the technical lag of workflow steps?* This question aims to quantify the extent to which workflows are outdated w.r.t. the Actions they are using. Based on an existing metric of *technical lag*, we show that most workflow steps are lagging behind the latest available release of the used Actions for at least seven months.

$RQ_6$ *What is the opportunity lag of workflow steps?* This question focuses on an alternative and complementary way of quantifying the extent to which workflows are outdated. By introducing and using a newly proposed metric of *opportunity lag*, we show that most workflow steps had the opportunity to update the Actions they use for nine months, but did not.

The remainder of this paper is structured as follows. Section 2 presents related work. Section 3 introduces the core concepts of GHA and the data extraction process. Sections 4 and 5 address the first and second research goal, respectively. Section 6 discusses the findings and their implications. Section 7 presents the threats to validity of the research, and Section 8 concludes.

## 2. Related work

### *2.1. On Continuous Integration, Deployment and Delivery*

Fowler and Foemmel proposed a set of 10 Continuous Integration (CI) core practices for software development acceleration and software quality enhancement in their seminal blog in 2000 [3]. Elazhary et al. [27] studied the benefits and challenges of these practices in three software-producing companies, observing that these practices were widely followed but with quite some variation. They called for more research to comprehend these variations and their effects on software quality and process improvement.

Vasilescu et al. [28] empirically analysed the effect of introducing CI to the pull request (PR) process of 246 GitHub projects. They found that CI improves productivity, leading to more PRs being processed, accepted and merged, without any negative side effect on code quality. Hilton et al. [29] studied the usages, costs and benefits of CI. Through a mixed-methods study involving 442 developers, 34K GitHub projects and 1.5M CI builds, they analysed which CI services developers use, and how and why developers use them. Among others, they found that most projects relied on Travis to implement their CI, and projects with CI tend to release twice as often, accept PRs faster, and their developers are less worried about breaking builds.

Shahin et al. [30] carried out a Systematic Literature Review (SLR) of 69 scientific articles to categorize the tools, approaches, challenges and practices related to continuous integration, delivery and deployment (CI/CD). They reported that CI/CD is mostly used to reduce build and test time, to increase the visibility and awareness of build and test results, to detect violations, and to improve the reliability of the deployment process. Soares et al. [31] also conducted an SLR on the use of CI. They analysed 101 empirical studies published between 2003 and 2019 to identify and interpret empirical evidence regarding how CI impacts software development activities and processes. They observed an increase in CI research in recent years, and much of the existing research reveals that CI brings positive effects to software development, such as improved productivity and efficiency, increased developer confidence, faster iterations and more stability. None of the aforementioned SLRs considered publications published after 2019, hence they do not include any study reporting on the use or impact of GHA.

Golzadeh et al. [4] published the first empirical study on CI services that also considered GHA. They conducted a longitudinal quantitative analysis on the use of 20 different CI services from 2011 to 2021 in 91K+ GitHub repositories related to npm packages. They observed that more and more repositories are relying on a CI service, and one out of two repositories used a CI in 2021. While Travis has been the dominant CI for years, the introduction of GHA in 2019 drastically changed the CI landscape on GitHub. It took only 18 months for GHA to become the most widely used CI service on GitHub, being used by more than half of the repositories with a CI. Rostami Mazrae et al. [32] confirmed these quantitative findings through a qualitative study on the usage, co-usage and migration of CI/CD tools based on in-depth interviews with 22 experienced practitioners. Among others, they observed a migration towards GHA due to its deep integration into GitHub, its generous free tier, its build support for the major operating systems, and its support for reusable Actions.

### *2.2. On GitHub Actions*

Due to the relative recency of GHA, research literature on the topic is still scarce. Kinsman et al. [33] analysed the impact of GHA in 3,190 repositories and observed that the adoption of GHA increases the number of rejected PRs and decreases the number of commits in merged PRs. Through a manual inspection of 209 issues related to GHA, they concluded that developers have an overall positive perception of GHA. Chen et al. [34] confirmed these observations in a replication study on 6,246 repositories.

Valenzuela-Toledo and Bergel [35] investigated the use and maintenance of GHA workflows in 10 popular GitHub repositories based on a manual inspection of 222 commits related to workflow changes. They determined 11 different types of workflow modifications and uncovered a number of deficiencies in workflow production and maintenance. This calls for adequate tooling to support creating, editing, refactoring, and debugging workflow files.

Saroar and Nayebi [36] surveyed 90 Action developers and users to understand their motivation, decision criteria, best practices and challenges in creating, publishing and using Actions. Among others, they found that Actions are hard to test and debug, and that security concerns are one of the five major challenges for developers when automating workflows on GitHub. Benedetti et al. [37] proposed a security assessment methodology to analyse the effects of security issues on GHA workflows and the software supply chain. They developed and applied the methodology through an automation tool on 50 GitHub projects, allowing them to uncover 25K security issues in workflows.

Decan et al. [6] analysed nearly 70K GitHub repositories in order the gain a deeper insight into the GHA ecosystem. They found that 43.9% of the repositories define a GHA workflow, and they characterised these repositories and their workflows in terms of which jobs, steps, and reusable Actions were used and how. They notably observed that workflows are primarily used for development purposes, despite the fact that many other kinds of activities could potentially be automated with GHA. They also observed that nearly all workflows use Actions, which may be problematic since issues in these Actions (e.g., bugs, security vulnerabilities, outdated or obsolete components) can propagate to the workflows that use them, potentially affecting the entire ecosystem.

The current paper naturally complements and extends this prior work by analysing the use ($\mathbf{G_1}$) and outdatedness ($\mathbf{G_2}$) of workflows w.r.t. the Actions they use.

### *2.3. On outdatedness of reusable software components*

It is common practice for software developers to depend on reusable software components to take advantage of ready-to-use code instead of developing everything from scratch [38]. To further facilitate this reuse, package managers and registries of reusable libraries have been proposed for the main programming languages (e.g., npm for JavaScript, PyPI for Python, Maven for Java). The wide availability and popularity of such reusable components facilitates building software, but it also causes software maintenance and evolution problems [7].

For example, even very recent versions of software application may be outdated because they depend on reusable components that were not updated to their latest versions.

Kula et al. [23] carried out an empirical study on software library migration in around 4.6K GitHub projects and 2.7K library dependencies. They found that 81% of the projects do not update their outdated dependencies. When surveying the project maintainers they discovered that most of them were actually unaware of such outdated dependencies. Mirhosseini and Parnin [39] studied software developers' incentives to update their project dependencies. They analysed 7.5K GitHub projects using badge notifications and automated PRs to inform developers about outdated dependencies and to update them. They found these mechanisms to be effective in encouraging developers to update their outdated dependencies, even though they could be improved further to better align with developers' expectations.

Keeping dependencies outdated incurs a higher risk of having bugs and security issues that may already have known fixes. Cox et al. [25] analysed 75 Java projects that manage their dependencies through Maven, and observed that projects relying on outdated dependencies were four times more likely to have security issues and incompatibilities. Lauinger et al. [19] published a comprehensive study on the security implications surrounding JavaScript library usage in 133K web applications. 37% of them were found to suffer from at least one security vulnerability due to an outdated dependency. Decan et al. [11] analysed to which extent security vulnerabilities propagate to npm packages through dependencies. They found that the use of dependency constraints plays an important role in fixing vulnerabilities due to dependencies: too restrictive dependency constraints prevent 40% of the releases to automatically benefit from security fixes deployed in more recent versions of their vulnerable dependencies.

Updating dependencies to more recent releases of reusable components is not for free since it might lead to backward incompatible changes. Moreover, making the right decision about choosing a proper version can be tricky and time-consuming [40]. Bavota et al. [41] studied the evolution of dependencies in Apache. They highlighted that the number of dependencies on reusable components is growing exponentially and must be taken care of by developers. They found that developers were reluctant to upgrade the libraries they depend upon because of breaking changes. Decan et al. [8] studied the use of dependencies in npm, CRAN and RubyGems. They found that most projects in these registries rely on other libraries and that there is an increasing number of projects whose failure can impact a wide number of (transitively) dependent projects. They also observed that, in combination with semantic versioning (SemVer[6]), dependency constraints can prevent packages from breaking due to dependency updates, while benefiting from bug and security fixes.

Raemaekers et al. [42] investigated to which extent SemVer is followed in more than 22K Java libraries on Maven. Breaking changes were observed in

[6]`https://semver.org`

one third of all releases, including presumably backward compatible patches and minor releases. Decan et al. [14] empirically studied the degree of SemVer compliance in the Cargo, npm, Packagist and RubyGems registries. They observed that compliant dependency constraints increased over time while registry-specific notations, characteristics, maturity and policy changes played an important role in the degree of such compliance.

Given the wide availability and use of Actions, GHA bears many similarities to these registries of reusable software libraries. Therefore, in this paper, we aim to quantify how frequently reusable Actions are updated ($RQ_2$), what are the versioning practices being followed by these Actions ($RQ_3$), and how frequently workflows are outdated with respect to the used Actions ($RQ_4$).

## *2.4. On technical lag*

The concept of *technical lag* in a software system was originally introduced by Gonzalez-Barahona et al. [43] to quantify how outdated a deployed software component is, reflecting "*the increasing lag between upstream development and the deployed system when no corrective actions are taken*". Technical lag is not only useful for *deployers* of software components, but also for the *developers* of such components. They can use technical lag to decide whether and when to update the dependencies of the components they maintain. That way, they can assess on an informed basis the risks of relying on outdated dependencies [25].

Zerouali et al. [44] used the concept of technical lag to empirically analyse package dependency updates in the npm registry. Their results indicate a strong presence of technical lag, with a median time gap of 3.5 months between the deployed version of a dependency and its latest available one. Decan et al. [17] analysed the evolution of the technical lag of 8M+ dependencies between 1.4M+ releases of npm packages. They found that one out of four dependencies suffers from technical lag, and around half of the updates are not automatically accepted because of the dependency constraints. They also evaluated that strict compliance to SemVer would reduce the exposure to technical lag by 18%.

Technical lag can be quantified along different dimensions: time (how old is a dependency compared to a more recent version?), version (how many versions is a dependency behind?), security (how many security vulnerabilities could have been fixed by updating the dependency?), and so on. Zerouali et al. [18] captured these dimensions in a conceptual measurement framework that can easily be applied to registries of reusable libraries. In particular, they analysed the technical lag of 500K+ npm packages, observing that around 26% of the dependencies in these packages are outdated and that half of these outdated dependencies target a release that is 270+ days older than the newer one. Zerouali et al. [45] also applied this technical lag framework to the DockerHub registry of Docker container images. They assessed the technical lag of 140K+ Debian-based container images considering five dimensions: the number of outdated packages in these images, the time difference between package releases, the number of missed releases, the number of vulnerabilities and the number of bugs.

Gonzalez-Barahona [46] proposed a model based on the technical lag framework to better understand the factors that influence outdatedness of an application produced with reusable components. He found that some factors depend on the upstream developers, on the application developers, and on the collection used to pull components from. Stringer et al. [47] carried out a large-scale analysis of technical lag across 14 major package managers. They found that technical lag is common in these package managers, although the quantity varies widely. They observed that technical lag is mostly due to strict dependency constraints, and that more permissive constraints induce less lag. For instance, they evaluated that a strict adherence to SemVer would eliminate a third of the technical lag. Based on these insights they advocate the use of automated dependency tracking and updating tools to expedite updates and minimise technical lag.

To reach goal $\mathbf{G_2}$ of this paper, we will use technical lag to assess to which extent reusable Actions used in workflows in GitHub repositories are outdated with respect to their latest available release ($RQ_5$). We will also propose and evaluate a complementary metric, baptised *opportunity lag*, to quantify how long a reused Action has had the opportunity to be updated but was not ($RQ_6$).

## 3. Methodology

### *3.1. About GitHub workflows and reusable Actions*

To enable GHA on a repository, one has to create one or more YAML files, each describing a single *workflow*, and store them in the `.github/workflows` folder. Figure 1 shows an example of a workflow file.[7]

A workflow defines a set of *events* (e.g., on lines 3-6, a push, a pull request and a scheduled event) that trigger the execution of a set of *jobs* (e.g., line 8). A job defines a list of *steps* (e.g., lines 12, 13, 17 and 19) that will be sequentially executed. Steps are the smallest units of work in a workflow. Through the `run:` key a step can specify the commands that will be executed (e.g., lines 18 and 20). A step can also delegate this task by calling a predefined reusable *Action* through the `uses:` key (e.g., lines 12 and 14).

An Action implements a single task (e.g., checking out repositories, deploying environments) that can be shared for reuse on a public GitHub repository and promoted on the GitHub Marketplace.[8] Workflow maintainers can use Actions in workflow steps to avoid having to write explicitly the various commands that need to be executed. Actions are powerful reusable components, since they 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).

When a step uses an Action, it should specify the name of the repository hosting the Action (e.g., actions/checkout). It can also specify the version of

[7]See `https://github.com/pandas-dev/pandas/blob/68f763e7/.github/workflows/code-checks.yml` for a more elaborate example of a workflow file.

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

```
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 dependencies
        run: pip install pytest
      - name: Execute tests
        run: pytest
```

Figure 1: Example of a GHA workflow file.

the Action that should be executed, by means of a step *anchor* (e.g., `@v2`). These anchors can reference a commit hash (e.g., `@753c60e0...`), a git tag (e.g., `@v2.1.3`), or even a branch name (e.g., `@main`).[9] This flexibility in how to specify anchors for steps is at the same time a source of confusion.

On the one hand, GitHub recommends Action maintainers to number new releases using semantic tags based on a three-component based versioning scheme (e.g., `v1.2.3`) in order to be compatible with the recommended practices of SemVer adopted by the npm JavaScript ecosystem. [10] In prior work we have analysed the benefits of adhering to such SemVer practices in software packaging ecosystems [14].

On the other hand, GitHub recommends workflow maintainers to refer to Action releases through their unique commit hash as it is "*the safest for stability and security*"[11] since "*it is currently the only way to use an action as an immutable release [and] helps mitigate the risk of a bad actor adding a backdoor to the Action's repository*".[12] This viewpoint that pinning Action dependencies can help to reduce security risks is shared by Gonzaga [48], member of the Node.js Technical Steering Committee. Because of this risk, the Node.js Security Working Group plans in 2023 to pin all Node.js Actions by their commit

[9]`https://docs.github.com/en/actions/creating-actions/about-custom-actions`

[10]`https://docs.github.com/en/actions/creating-actions/releasing-and-maintaining-actions`

[11]`https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions`

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

hash.

The interested reader is invited to consult the official GitHub documentation[13] or Chandrasekara and Herath's book [5] to obtain more information about GHA.

## *3.2. Data extraction*

To conduct an empirical study on the outdatedness of GHA in software development repositories, we need a large dataset of GitHub repositories, excluding repositories that are used only for experimental or personal reasons, or that show no or little traces of actual software development activity [49]. We relied on the SEART GitHub search engine [50] to obtain a list of candidate repositories. We queried the tool on 2022-09-05 to get all non-fork repositories that were created before 2022, which were still active in 2022, and had at least 100 commits and 100 stars. We obtained 62,673 repositories satisfying these criteria.

On 2022-09-05, we locally cloned these repositories to look for the presence of workflow files in the `.github/workflows` folder of the default branch reported by the GitHub API. We found 22,758 repositories (i.e., 36.3%) that contained YAML files in this folder. For each repository, we used the `git rev-list` command-line tool to obtain the last commit of each month between October 2019 and August 2022, hence representing the state of the repository for the first day of each month between November 2019 and September 2022. We parsed the contents of the YAML files in these commits using the ruamel.yaml Python library to check whether they actually define a GHA workflow. If that was the case, we extracted the relevant workflow data (e.g., workflow name, events triggering the workflow), the jobs configured in the workflow (e.g., job name) and the steps defined in these jobs (e.g., step name, commands associated to `run:` key, Actions associated to `uses:` key). By means of a regular expression, we extracted for each step relying on an Action its provider (i.e., the owner of the repository providing the Action), the Action name and the *anchor* used to refer to that Action.

The dataset resulting from this process contains 22,758 distinct GitHub repositories with 979,591 workflows and 7,927,295 steps over 35 monthly snapshots from November 2019 to September 2022. Figure 2 shows the evolution through these 35 monthly snapshots of the number of repositories using GHA, as well as the number of workflows and steps. The last considered snapshot contains 53,801 workflows including 330,186 steps. The figure also reveals slight variations for the end of 2020. They coincide with restrictions imposed by Travis on its free plan for public repositories, leading many repositories to switch from Travis to GHA [4].

The data and code to replicate the analysis are available on `https://zenodo.org/record/8073333`.

[13] `https://docs.github.com/en/actions`

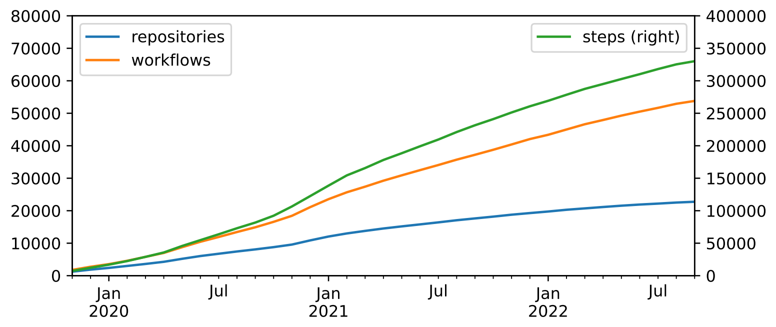

Figure 2: Evolution of the number of repositories using GHA and their number of workflows (left y-axis) and number of steps (right y-axis).

## 4. Research goal $\mathbf{G_1}$

The aim of $\mathbf{G_1}$ is to provide quantitative insights in the prevalence of the use of Actions in GitHub repository workflows ($RQ_1$), as well as their evolution over time in terms of update frequency ($RQ_2$) and versioning practices used ($RQ_3$). This is a prerequisite for goal $\mathbf{G_2}$ since it would not make much sense to study outdatedness of workflows w.r.t. their used Actions if it turns out that workflows are rarely using Actions or if Actions are rarely releasing new versions.

*$RQ_1$ To which extent do workflows rely on reusable Actions?*

This question aims to quantify to which extent steps, workflows and repositories are making use of Actions.

To identify steps using an Action, as explained in Section 3, we used a regular expression to match the pattern `uses: <provider>/<action>[@<anchor>]`, where `<provider>` corresponds to the owner of the repository in which the `<action>` is developed, and where the optional `<anchor>` denotes the version of the `<action>` that should be used in the step.[14] A typical example is `uses: actions/checkout@v2` which is the most frequently observed cases in our dataset, accounting for 26.1% of all matches.

Figure 3 shows the evolution of the proportion of repositories, workflows and steps that use an Action. We report on these three levels of granularity since a repository can contain multiple workflows, and a workflow can have multiple steps. We observe that the vast majority of steps are re-using an Action (through the `uses:` key), as opposed to defining sequences of commands (through the `run:` key). The proportion of such steps ranges from 61.7% to 64.3%, at the end of the observation period. As a consequence of this prevalent reuse of Actions in steps, nearly all workflows and repositories contain steps that are using an

[14]GitHub's documentation strongly recommends the use of an anchor. Yet, not specifying an anchor results in a build failure, explaining why only 25 steps in 4 repositories in our dataset do not specify an anchor.

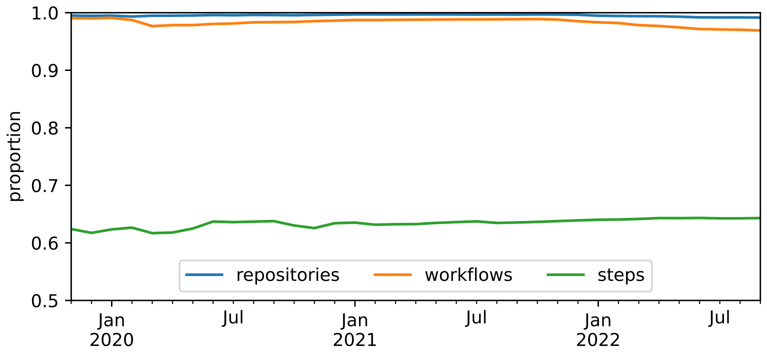

Figure 3: Evolution of the proportion of repositories, workflows and steps using an Action.

Action. In the latest considered snapshot, 99.7% of the repositories and 99.1% of the workflows have at least one step using an Action.

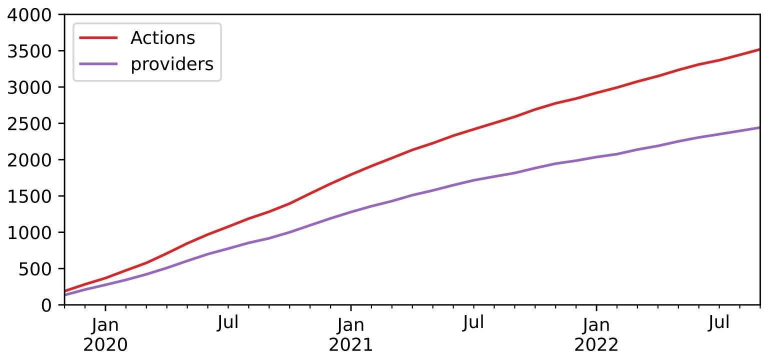

Figure 4: Evolution of the number of distinct Actions and distinct providers.

We identified how many and which Actions are used in workflows. Figure 4 reports on the number of distinct Actions and their providers. It shows a continuous growth in the number of distinct Actions being used in workflows. Over the entire observation period, we found 3,519 distinct Actions distributed by 2,442 distinct providers. On average, in the latest considered snapshot, a workflow makes use of 2.7 distinct Actions (median is 2), for a total of 2,930 distinct Actions from 2,064 distinct providers.

However, these 2,930 distinct Actions are not equally used by the steps and a very large number of steps are using a limited number of distinct Actions. For instance, the 23 most used Actions, accounting for only 0.79% of the Actions, already concentrate more than 80% of all reuse. Moreover, 12 Actions out of these 23 are provided by GitHub (those prefixed with actions/), and actions/checkout alone is used in 36% of the steps. The first most widely used Action not provided by GitHub is actions-rs/toolchain, which is used in 3,784 steps (i.e., 1.78% only).

In order to determine which providers supply the most frequently used Actions, we identified for the latest snapshot the steps, workflows and repositories using the Actions provided by them. Table 1 shows the top 10 providers along with the number of Actions they provide (in our dataset), and the proportion of steps, workflows and repositories using them. For the sake of completeness, we also provide these proportions for all *other providers*. Only steps, workflows and repositories making use of an Action are considered for this analysis.

Table 1: Top 10 providers whose Actions are the most widely used by steps in the latest considered snapshot.

| **provider** | **# Actions** | **% steps** | **% workflows** | **% repositories** |
|---|---|---|---|---|
| actions | 26 | 67.14% | 95.44% | 98.83% |
| docker | 7 | 5.06% | 4.81% | 7.64% |
| actions-rs | 7 | 2.19% | 3.71% | 4.45% |
| codecov | 1 | 1.63% | 4.40% | 9.20% |
| shivammathur | 2 | 1.10% | 2.92% | 4.37% |
| ruby | 2 | 0.94% | 2.53% | 4.26% |
| peter-evans | 15 | 0.63% | 1.57% | 2.60% |
| softprops | 2 | 0.61% | 1.64% | 3.36% |
| peaceiris | 5 | 0.53% | 1.33% | 2.93% |
| pypa | 2 | 0.52% | 1.29% | 2.77% |
| *other providers* | 2,861 | 19.65% | 38.23% | 48.33% |

As expected from the previous analysis, actions/ is the major provider of Actions used in steps, workflows and repositories. These Actions proposed by GitHub itself correspond to the *basic building blocks* for workflows. For instance, actions/checkout allows to clone the current repository efficiently, Actions of the form actions/setup-x setup the corresponding programming language environments (e.g., Java, Python or npm), actions/cache is used to cache dependencies and build outputs to improve workflow execution time, whereas actions/upload-artifact and actions/download-artifact are used to share artifacts between jobs during the execution of a workflow. As such, it is not that surprising that the Actions belonging to this provider are used in 67.14% of the steps, 95.44% of the workflows, and 98.83% of the repositories in the latest considered snapshot.

Table 1 also reveals that the Actions provided by docker/ are the second most widely used in the latest snapshot, occurring in 5.06% of the steps, 4.81% of the workflows and 7.64% of the repositories. Examples of widely used Docker Actions are docker/login-action to login against a Docker registry, docker/build-push-action to build and push Docker images, docker/setup-buildx-action to setup a Docker buildx environment, and docker/setup-qemu-action to setup QEMU, an open source machine virtualizer.

Providers not belonging to the top 10 of Table 1 are still contributing widely-used Actions. They account for 2,861 Actions (compared to only 69 Actions coming from the top 10 providers), and are used by 19.65% of the steps in nearly half of the repositories (48.33%) in the latest snapshot.

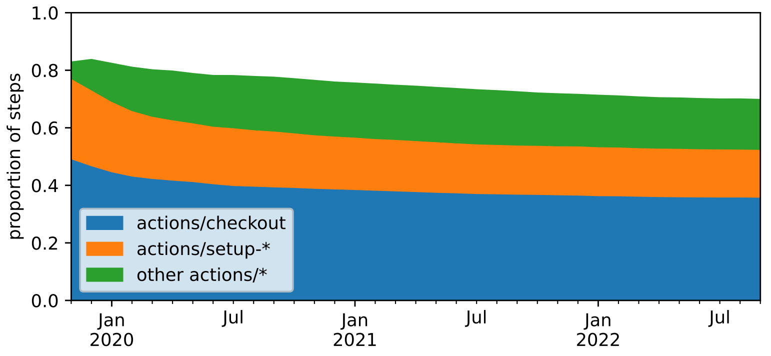

Figure 5: Evolution of the proportion of steps using Actions from the actions/ provider.

Since the actions/ provider accounts for the majority of the most widely used Actions, we quantified the evolution of the proportion of steps using these Actions. It should be noted that, over time, we observed in our dataset 32 different Actions belonging to the actions/ provider, of which 26 are still being used in the latest considered snapshot. Figure 5 shows the evolution of the proportion of Actions provided by actions/, distinguishing between the most widely used actions/checkout, the 8 Actions of the form actions/setup-x (e.g., setup-node or setup-python), and the 23 remaining Actions provided by actions/ (e.g., cache, upload-artifact and download-artifact).

We also observe that from 83.6% (December 2019) to 69.7% (September 2022) of the steps are using Actions from the actions/ provider, the checkout Action being used by 36% of the steps in the last snapshot, the 8 Actions of the form actions/setup-x by 16.6% of the steps, and the 23 remaining Actions from actions/ by 17.1% of the steps. This confirms the uneven distribution of Action reuse in steps, mostly concentrated on a few Actions provided by GitHub [6].

**Summary:** Using Actions is common practice in workflows. More than 60% of the steps are using an Action. Nearly all repositories and workflows have at least one step using an Action. A few Actions concentrate most of this reuse. Actions provided by GitHub account for more than two thirds of this reuse.

*$RQ_2$ How frequently are Actions and workflows updated?*

In $RQ_1$ we found that relying on reusable Actions is common and that a limited number of Actions concentrate most of the reuse. $RQ_2$ aims to identify how frequently workflows and Actions are updated, which is a preliminary step needed to assess the outdatedness of workflows relying on Actions.

Detecting how frequently a workflow is updated is easy since our dataset contains monthly snapshots of the workflow files for all the considered repositories.

On the other hand, detecting when an Action is updated is more challenging. Since Actions are developed in public GitHub repositories, we have access to all the commits made to develop and update an Action. However, not all commits lead to a new distributable release of the corresponding Action. As a consequence, we relied on the GitHub API for "releases"[15] to detect when an Action has been updated. However, not all Actions use GitHub's release system: out of the 3,519 distinct Actions identified in $RQ_1$, we were able to find releases for 2,811 Actions (i.e., 79.9%), accounting for a total of 28,889 releases.

In the latest considered snapshot, the 708 Actions without known releases are used by 1,512 steps, 997 workflows, and 704 distinct repositories. Proportionally, this represents only 0.71% of the steps, 1.9% of the workflows, and 3.1% of the repositories making use of an Action, even though these Actions account for 20.1% of the distinct Actions we found in workflow steps.

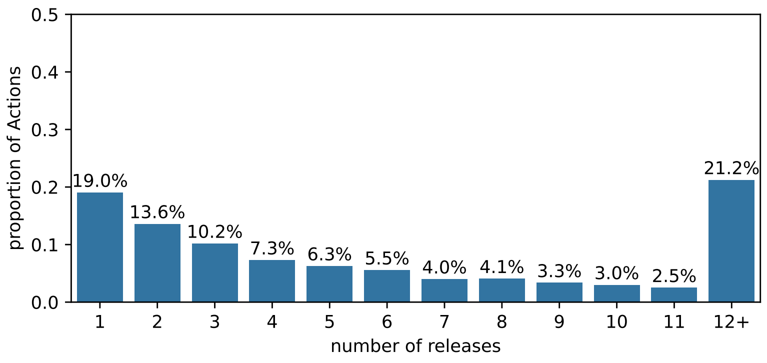

Figure 6: Proportion of Actions in function of their number of releases.

Based on this data, we found Actions to have 10.3 releases on average (2, 4 and 10 respectively for the 25th, 50th and 75th percentiles). Figure 6 shows the proportion of Actions in function of their number of releases. We observe that 19% of the Actions in our dataset have a single release, while on the other extremity 21.2% have 12 or more releases. Around half of all Actions have 5 or more releases. To assess the frequency of updating Actions we computed the time between consecutive releases. It takes, on average, 38 days to release a new version of an Action, and respectively 0.3, 5, and 32 days for the 25th (i.e., Q1), 50th (i.e., median) and 75th (i.e., Q3) percentiles.

For each monthly snapshot during the observation period we identified the workflows that were updated, by checking if their contents differed w.r.t. the snapshot of the preceding month. We also identified the Actions that were updated, by checking for new available releases during the considered month. Figure 7 shows the monthly proportion of Actions and workflows having been

[15] `https://api.github.com/repos/{owner}/{repo}/releases`

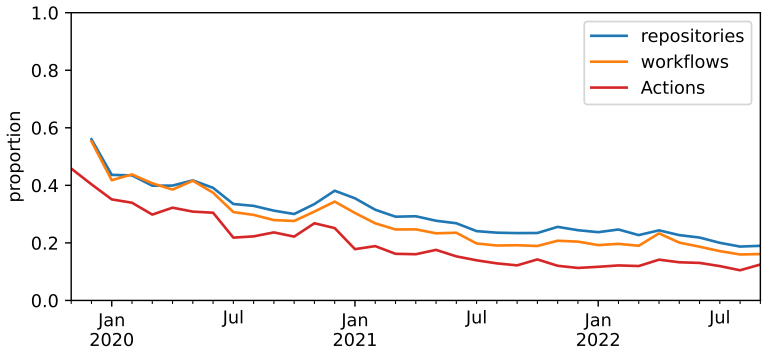

Figure 7: Monthly proportion of Actions and workflows having been updated and of repositories having updated a workflow.

updated as well as the monthly proportion of repositories having updated a workflow. We observe a decreasing trend for all curves. This is a consequence of an increase in the total number of considered Actions, workflows and repositories (see Figure 2 and Figure 4) combined with an increasing number of inactive Actions and workflows that are no longer updated. Each month, on average, 19.5% of the Actions are updated (median is 16.1%), 27.1% of the workflows are updated (median is 24.1%), and 32.1% of the repositories have updated a workflow (median is 29.1%). During the last observed month, 12.5% of the Actions released a new version, while 16.1% of the workflows were updated, and 19.0% of the repositories updated a workflow.

**Summary:** Actions and workflows are continuously updated. Half of the Actions have 5 or more releases. It takes 38 days on average to release a new version of an Action. Each month, one Action out of five and one workflow out of 4 are updated, and one repository out of three updates a workflow.

## *$RQ_3$ Which versioning practices are being used in GHA?*

*$RQ_2$* showed that Actions are frequently updated. As such, workflows making use of these Actions are prone to rely on older releases of these Actions. As explained in Section 3.1, when a workflow step uses a predefined Action, in addition to specifying the name of the repository providing the Action, it should use an *anchor* to specify which version of the Action should be selected. We also explained that the two recommended ways by GitHub to specify anchors are by using either a SemVer-compatible versioning scheme (e.g., `v2` or `v1.2.3`) implemented through git tags, or by relying on a unique immutable commit hash (e.g., `@753c60e0...`) to avoid malicious actors to override the specified version (which is possible with mutable git tags). These two recommendations

being contradictory, $RQ_3$ aims to identify which versioning practices are actually followed for Actions and step anchors.

We started by looking at how releases of Actions are specified. To do so, we manually inspected the releases of a few hundred randomly selected Actions to identify which versioning schemes were used. Based on this qualitative analysis three main categories naturally emerged: component-based version numbers (e.g., `v1.2` or `1.2.3-alpha`), date-based versions (e.g., `20210913`) and commit hashes (e.g., `2bccd0e0...`). We subdivided the category of component-based version numbers based on the number of components being specified. We defined regular expressions to match the different categories, and applied them on all Action releases of our dataset. All anchors that did not fit into one of the aforementioned categories (e.g., `latest` or `alpha`) were grouped together in the "other" category, which only contained 0.51% of the Actions (on average).

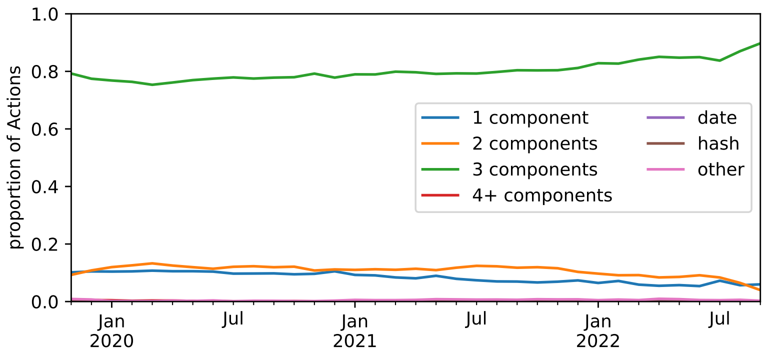

Figure 8: Evolution of the proportion of Actions in function of their versioning scheme.

Figure 8 shows the evolution of the monthly proportion of Actions in function of the identified versioning scheme. For each month and each Action, we classified its versioning scheme based on its latest version released during the month. We observe that the most widely followed versioning scheme is the component-based one. In the last month of the observation period, it is used by 99.7% of the Actions. In particular, the 3-component based notation is used by 89.7% of the Actions. This suggests that GitHub's recommendation to adopt a SemVer-based notation is widely followed.

We applied the same methodology to identify the versioning practices followed in workflows that use Actions in some of their steps. A manual investigation allowed us to identify the same categories of versioning schemes as before. The main differences were that date-based versions were not used in steps, and that the "other" category mostly contains branch names (e.g., `master` or `main`).

Figure 9 shows the monthly proportion of workflow steps using an Action, grouped by category of versioning scheme used by the anchor. We again observe that component-based notations are the most widely used, accounting for 92.7% of the anchors used in the latest snapshot. In contrast to the previous

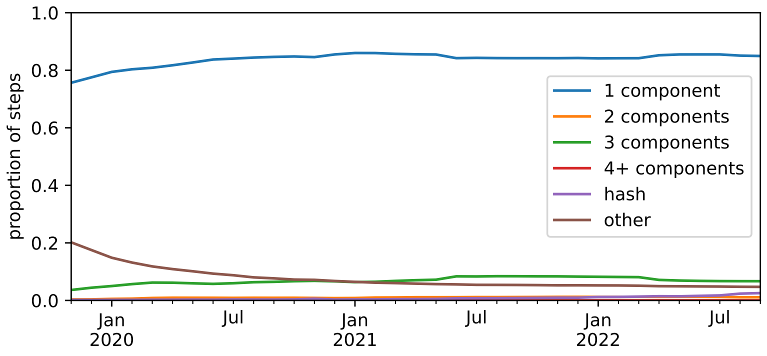

Figure 9: Monthly proportion of steps using an Action, grouped by the versioning category used in the anchor.

analysis, however, a minority of the steps are using the 3-component notation for their anchors (only 6.7% compared to 89.7% of the Actions that specified a 3-component version in their latest snapshot). Instead, the majority of the steps (85.0%) are using a 1-component notation (e.g., `@v3`) to refer to a release of an Action. These observations are in line with our observations in prior work [6], with the notable difference that in that prior work we used a less recent dataset focused on a single snapshot, and we did not distinguish between the number of components used to refer to an Action in workflow steps.

The discrepancy between how versions are specified in Actions and how they are used in steps is not really surprising. Indeed, assuming adherence to SemVer, GitHub recommends specifying the version tag in anchors by including only the major component (e.g., `@v2` instead of `@v2.1.3`) in order to receive critical fixes and security patches while still maintaining compatibility. At the same time, GitHub recommends maintainers of reusable Actions to manually create additional git tags for "*keeping major (v1) and minor (v1.1) tags current to the latest appropriate commit*". This explains why most of the steps are relying on a 1-component notation to refer to the 3-component versions of Actions. However, this implies that Action maintainers must *move* some of these git tags each time a new version of the Action is released (e.g., moving `@v2` and `@v2.1` tags from release `v2.1.3` to release `v2.1.4` when new version `2.1.4` is released). Unless automated, this introduces an additional burden on the maintainers. Forgetting to update these tags when an Action releases a minor or a patch update implies that the steps that depend on it do not automatically benefit from the bug and security fixes provided by the update.

Coming back to Figure 9, perhaps the most surprising and worrisome finding is that the most secure way of referring to the release of an Action, namely through immutable commit hashes, is used by only 2.6% of the steps in the latest snapshot, despite GitHub's security recommendation to do so (see Section 3.1). The use of commit hashes is currently the only way to enforce so-called *pinned dependencies*, which is one of the scoring criteria used by the Open Source

Security Foundation (OpenSSF) to assess insecure open source projects.[16] The Node.js Security Working Group is planning to adopt these security practices in 2023 [48], so we expect to see an increase in the proportion of steps using hash-based anchors in the near future.

> **Summary:** 9 out of 10 Actions use three-component versioning for their releases, while less than 1 out of 10 steps uses such notation to refer to an Action. 85% of all steps refer to Actions using single-component versioning, suggesting a large adherence to GitHub's recommendation to use a SemVer-compatible way to depend on Actions. GitHub's security recommendation to use commit hashes to pin dependencies to Actions used in workflow steps is rarely followed.

## 5. Research goal $\mathbf{G_2}$

In $\mathbf{G_1}$ we showed that relying on reusable Actions is common and that a limited number of Actions concentrate most of the reuse ($RQ_1$). We found that Actions are continuously updated ($RQ_2$) and that most Actions follow a three-component versioning notation and that GitHub's recommendation for semantic versioning is widely followed ($RQ_3$). This justifies the need to study a second goal $\mathbf{G_2}$, aiming to understand to which extent workflows become outdated in terms of the reusable Actions used by them ($RQ_4$), and quantifying this outdatedness in terms of two complementary metrics of technical lag ($RQ_5$) and opportunity lag ($RQ_6$).

*$RQ_4$ How prevalent are outdated workflows?*

The wide availability and reuse of Actions within workflows, combined with the frequent release of new versions of Actions, increases the likelihood of workflows becoming outdated. Therefore, $RQ_4$ aims to quantify how frequently workflows are outdated with respect to the Actions they are using.

In order to identify which workflow steps are outdated with respect to a used Action, we need to map the specified `anchor` to one of the releases available for the corresponding Action. However, as observed in $RQ_3$, there is a discrepancy between how steps refer to Action versions and how Action releases specify their versions. This discrepancy implies there is no one-to-one mapping between the Action versions specified in steps and the versions specified in Action releases. Indeed, steps usually refer to the major version of an Action by means of a 1-component version number, while Action releases usually specify a 3-component version number. Action maintainers have to define additional git tags with 1-component version numbers as aliases for the latest corresponding release (e.g., git tag `v2` would be introduced as an alias for latest version `v2.1.3`, and this

[16] `https://securityscorecards.dev`

alias would be updated to the next latest version `v2.1.4` whenever it becomes available).

Unfortunately, there is no historical data about git tags since the git versioning system does not store this in its history log. This makes it impossible to identify which commit (and hence, which release) was targeted by a given git tag at a given point in time. To circumvent this intrinsic limitation of git we applied the following procedure. We first looked for an exact match between the version specified in the anchor and one of the releases available in the corresponding snapshot of the targeted Action. If no exact match could be found, and if the version category of the anchor was "component-based", we identified the highest available release by means of prefix matching (e.g., `@v2` and `@v2.3` would both be mapped to `v2.3.4`). This approach, whose threats to validity are discussed in Section 7, allows us to simulate GitHub's recommendation for SemVer under the assumption that the corresponding git tags are created and updated appropriately. Using this procedure we were able to map the versions specified in anchors for 3,422,135 steps, corresponding to 92.6% of the steps that were using an Action with known releases. Out of these steps, 347,510 (i.e., 10.15%) were mapped with an exact match and the remaining ones with the above prefix-based matching approach.

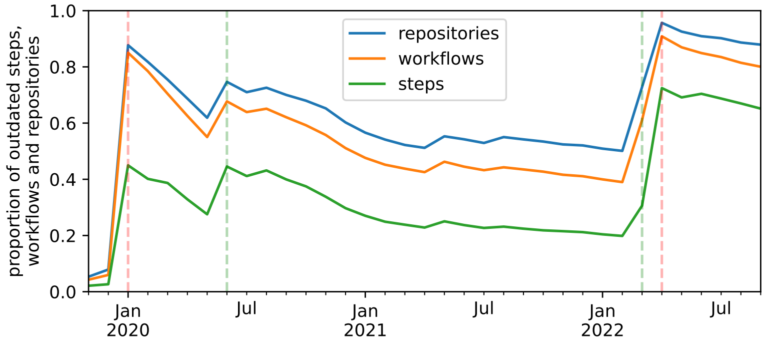

Figure 10: Monthly proportion of repositories, workflows and steps using an outdated release of an Action.

Figure 10 shows the monthly evolution of the proportion of repositories, workflows and steps using an outdated release of an Action. We observe that all curves exhibit a similar behaviour, with an average difference of around 21% between steps and workflows, and of around 8% between workflows and repositories. The proportion of outdated workflows (resp. steps) slowly decreased from 85% (resp. 45%) in January 2020 to 39% (resp. 20%) in February 2022, before jumping to 91% (resp. 72%) in April 2022. In the latest snapshot, 65.2% of the steps, 80% of the workflows and 87.9% of the repositories are using an outdated release of an Action.

Over the considered observation period, we observe 4 sudden increases in the proportion of outdated steps, workflows and repositories, indicated by vertical dashed lines. We manually looked at the steps that led to these increases and found that the two increases indicated by a vertical red dashed line (in January 2020 and April 2022) correspond to the release of a new major version of actions/checkout (respectively, v2 and v3). Similarly, the two increases indicated by a vertical green dashed line (in June 2020 and March 2022) correspond to the release of new major versions (v2 and v3) for the Actions of the form actions/setup-x. It is not surprising that major new releases for these Actions have a huge impact on the proportion of outdated repositories and steps, since Figure 5 revealed that these Actions account together for 52.6% of the reuse (in the latest snapshot).

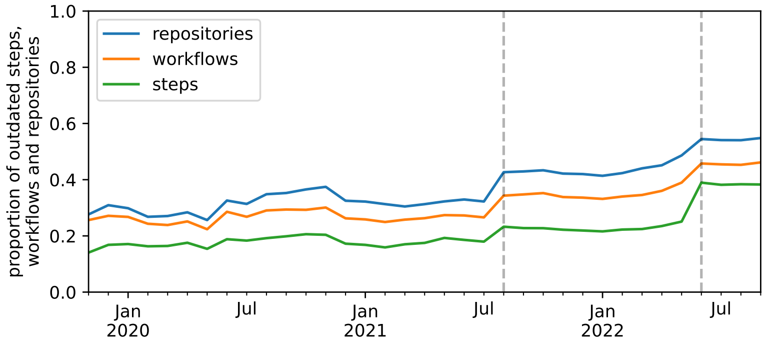

Figure 11: Monthly proportion of repositories, workflows and steps using an outdated release of an Action, excluding steps using Actions from the actions/ provider.

Since the few Actions from the actions/ provider somehow "hide" the impact of 2K+ other Actions used by steps, we repeated the analysis by excluding steps using Actions from the actions/ provider. Figure 11 reports on the new proportions, revealing that the proportion of outdated workflows (resp. steps) is slightly increasing through time, going from 25.6% (resp. 14.1%) at the beginning of the observation period to 46% (resp. 38.3%) in the latest snapshot. Most of this increase can be explained by two distinct events in August 2021 and June 2022, indicated by vertical dashed lines. The first increase was due to release 2.0.0 of codecov/codecov-action, leading 1,413 additional steps to become outdated. The second increase was due to various Actions from the docker/ provider that were updated nearly at the same time. For instance, release v3.0.0 of build-push-action, and release v2.0.0 of login-action, setup-buildx-action and setup-qemu-action caused 6,059 additional steps to become outdated.

**Summary:** Nearly two thirds of the steps and four out of five workflows are using an outdated release of an Action. Steps using Actions provided by GitHub are responsible for most of the outdatedness. Excluding these steps, more than one third of the steps and nearly half of the workflows are using an outdated release of an Action.

*$RQ_5$ What is the technical lag of workflow steps?*

In $RQ_4$, we showed that most of the workflows are using an outdated release of an Action. With $RQ_5$, we aim to quantify the extent to which workflows are outdated w.r.t. the Actions they are using. To do so, we rely on the concept of *technical lag*, which quantifies the outdatedness of a *(re)used* version of a software component by comparing it with an *ideal* version of this component (e.g., a more recent, more stable or less vulnerable version). This concept was formalised into a technical lag measurement framework for reusable software components by Zerouali et al. [18], presenting how technical lag can be quantified in terms of time, versions, bug or security issues, etc. $RQ_5$ applies this framework to compute, for all monthly snapshots, the technical lag of workflow steps that (re)use Actions.

More specifically, we compute technical lag as *the time difference (expressed in months) between the release of an Action used by the step and the latest available release of this Action in the considered snapshot.*[17] Intuitively, this corresponds to the number of months of development effort that is "missed" by using an outdated release of an Action rather than the more recent one.

Table 2: Example of technical lag for steps in the `test.yml` workflow from the acm309/putongoj repository, taken on 1 September 2022.

| | **uses: specification of step** | **selected** | **latest** | **technical lag** |
|---|---|---|---|---|
| 1 | actions/checkout@v3 | v3.0.2<br>22-04-21 | v2.4.2<br>22-04-21 | *up-to-date* |
| 2 | actions/setup-node@v3 | v3.4.1<br>22-07-14 | v3.4.1<br>22-07-14 | *up-to-date* |
| 3 | supercharge/redis-github-action@1.4.0 | 1.4.0<br>21-12-28 | 1.4.0<br>21-12-28 | *up-to-date* |
| 4 | supercharge/mongodb-github-action@1.7.0 | 1.7.0<br>21-11-16 | 1.8.0<br>22-08-26 | 9.4 months |
| 5 | pnpm/action-setup@v2.0.1 | v2.0.1<br>21-04-24 | v2.2.2<br>22-05-28 | 13.3 months |
| 6 | codecov/codecov-action@v2 | v2.1.0<br>21-09-13 | v3.1.0<br>22-04-21 | 7.3 months |

Table 2 shows an analysis of the technical lag of the `test.yml` workflow in the acm309/putongoj repository taken on 1 September 2022. This workflow defines 8 steps, of which 6 are relying on an Action. The first two belong to the actions/ provider and are referred to using the `@v3` anchor, which points to the highest available release of the corresponding Action, implying that they do not have any technical lag. It is interesting to note that the *highest* version `v3.0.2` of actions/checkout was released shortly before the *latest* version `v2.4.2` that essentially corresponds to a backport of the changes made in `v3.0.2`.

[17]Since backporting bug and security fixes to previous major branches is common practice in ecosystems of reusable software components [15], we select the *highest* available release instead of the *latest* one for Actions adhering to component-based versioning (e.g., version `v3.0.2` is preferred to `v2.4.2` even if the latter was released later).

One step (number 6 in Table 2) refers to codecov/codecov-action@v2 which points to an outdated release `v2.1.0`, implying that the used Action version has a technical lag of 7.3 months with respect to the latest available release `v3.1.0`. Three other steps (numbers 3, 4 and 5 in Table 2) use some Action through a 3-component anchor. In two of these cases, the corresponding Action release is outdated since a higher version has been released more recently, leading to a technical lag of 9.4 and 13.3 months, respectively.

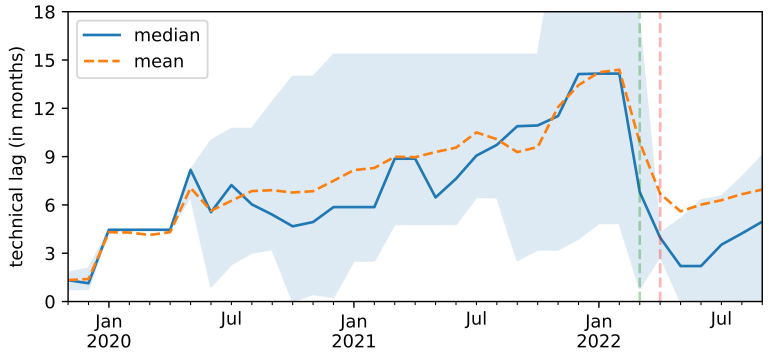

Figure 12: Evolution of the distribution of technical lag for outdated steps. The shaded area corresponds to the interval between the 25th and 75th percentiles.

Figure 12 shows the monthly evolution of the distribution of technical lag for outdated steps. The blue line indicates the median technical lag expressed in months while the dashed orange line indicates the mean lag value. The shaded area corresponds to the interval between the 25th and 75th percentiles. We observe that the median and average technical lag value is steadily increasing from around 3 months to around 14 months in March 2022. At the end of the observation period, the median and mean technical lags are of 5.0 and 7.0 months, respectively.

We also observe a strong decrease in March, April and May 2022, corresponding to the release of a new major `v3.0.0` version for the various actions/setup-x Actions in March 2022 (indicated by a vertical dashed green line), and for actions/checkout in April 2022 (indicated by a vertical dashed red line). While one may expect the technical lag to increase when new versions are released, especially in the case of a major version, we observe the opposite here, for two different reasons:

1. The first reason is that we report on the distribution of technical lag for steps that are *outdated* only. The new major releases led a plethora of steps using them with a `@v2` anchor to start contributing to the distribution of technical lag with a “low” lag value. For example, 17,052 steps using one of the actions/setup-x Actions started to contribute to the distribution of technical lag with an average value of 1.2 months, whereas the distribution was driven so far by 34,327 other steps with an average of 14.2 months.

2. The second reason is that new versions for the `v2` branch were released briefly after `v3.0.0`, incidentally leading the technical lag of steps using `@v2` to decrease. For example, versions `v2.4.1` and `v2.4.2` of actions/checkout were released in April 2022, nearly at the same time as versions `v3.0.1` and `v3.0.2`. This led the technical lag of 51,223 steps to lower from 4 months to less than a day for the snapshot of May.

Since most of the technical lag is explained by steps using Actions from the actions/ provider, we repeated the analysis by excluding the steps using these Actions. Figure 13 shows a continuously increasing technical lag for the remaining steps, going from an average of 1.3 month up to 9.8 months in May 2022. In the latest considered snapshot, the median and mean technical lag are of 7.3 and 8.3 months, respectively. This means that half of the outdated steps are using an Action version that is outdated w.r.t. the latest one for at least 7.3 months. A quarter of the outdated steps even lag behind for more than a year.

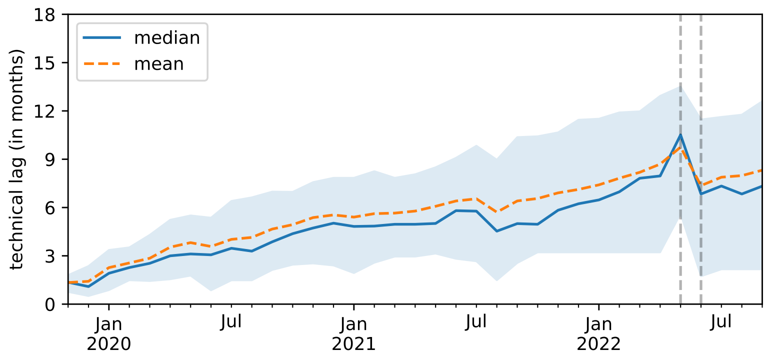

Figure 13: Evolution of the distribution of technical lag, excluding outdated steps using Actions from the actions/ provider. The shaded area corresponds to the interval between the $25^{\text{th}}$ and $75^{\text{th}}$ percentiles.

We also observe a sudden increase in May 2022 followed by a slightly higher decrease in June 2022, indicated by the vertical dashed lines. A manual inspection revealed that these variations are due to two separate and unrelated events. The increase in May 2022 is mostly due to the new `v3.0.0` and `v3.1.0` releases of codecov/codecov-action (see also the example presented in Table 2). While most of the steps using this Action were already outdated at that time, these newer releases further increased the technical lag from an average of 4.5 months to 10.1 months. The decrease we observe in June 2022 is a consequence of the docker/ provider having released a new major version for several of its widely used Actions, as explained in $RQ_4$. This led the number of steps contributing to the technical lag distribution to increase from 285 to 6,344. However, at the same time, their average induced lag decreased from 12.8 to 3.8 months, explaining why the overall distribution exhibits a decreasing trend.

**Summary:** Technical lag of outdated steps is continuously increasing. Most of the technical lag is explained by outdated steps using Actions provided by GitHub. Half of the outdated steps using other Actions are using a version that is lagging behind the latest one for at least 7.3 months.

*$RQ_6$ What is the opportunity lag of workflow steps?*

The technical lag of Actions, as used in $RQ_5$, helps to quantitatively assess the extent to which steps are outdated with respect to the used Actions, by comparing how long an Action version used in a step lags behind the latest or highest available version. A high technical lag may signal workflow maintainers the need to update the Action versions used in steps, assuming this is possible.

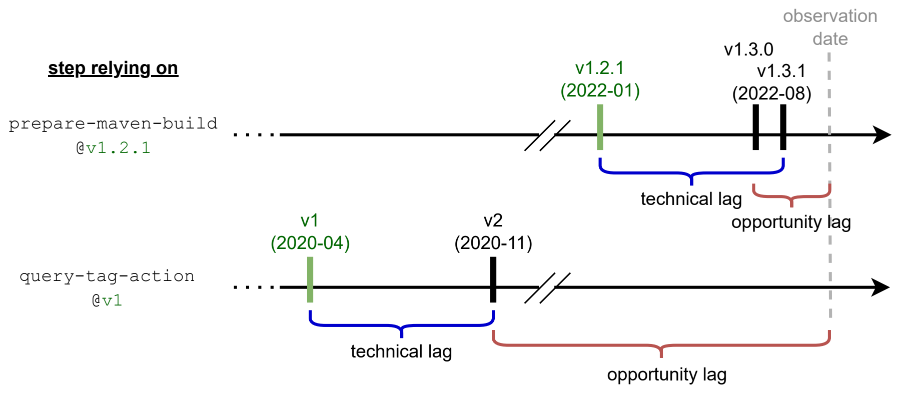

Figure 14: Illustration of the technical lag and opportunity lag for two outdated steps.

However, technical lag does not take into account the time during which such an update could have been adopted, and can therefore remain high regardless of how long a newer release of the Action has been available (which could range from a couple of days to several years). Consider for example the two situations illustrated in Figure 14, on the observation date of 1 September 2022. In the first situation, workflow `release.yml` in repository schemacrawler/schemacrawler contains the instruction `uses: sualeh/prepare-maven-build@v1.2.1` to rely on some Action. Version `1.2.1` of this Action was released in January 2022, and it remained the latest available release until 31 August 2022, when two new versions `1.3.0` and `1.3.1` were released. These new versions induce a technical lag of nearly 8 months even though they were released only a single day before the observation date of 1 September 2022. This means that the workflow maintainer only had one day available (w.r.t. the observation date) to seize the opportunity to update the workflow to reduce its lag.

The second situation shows the `package-beta.yml` workflow of repository xyoye/dandanplayforandroid. This workflow uses jimschubert/query-tag-action@v1. Version `v1` of this Action was released in April 2020 and remained the latest available release until November 2020, when version `v2` was released. The technical lag for this step on the observation date of 1 September 2022 is therefore nearly 8 months as well, as in the first situation. The main difference, however, is that the new release has been available for 22 months, leaving plenty of time for the workflow maintainer to seize the opportunity of updating the step to refer to this newer release.

Even though the technical lag is roughly the same in both situations, they drastically differ in the timeframe during which the workflow maintainers had the opportunity to update the outdated steps. To capture the difference between both situations, we introduce a variant of technical lag, called ***opportunity lag***, to quantify the time period during which a workflow maintainer could have updated an outdated step to a more recent version of an Action. More specifically, we compute the opportunity lag for an outdated step at time $t$ *as the difference in time between t and the first release of the Action that caused the step to become outdated.* According to this definition, the opportunity lag is one day in the first situation, and 22 months in the second situation.

From a pure definition point of view the *technical lag* ($RQ_5$) and *opportunity lag* ($RQ_6$) capture different aspects of step outdatedness, and the previous example illustrates their complementarity. In order to determine whether both metrics are actually correlated, we computed both variants of lag measurement for all the outdated steps in the latest consider snapshot. We excluded steps using Actions from the actions/ provider for the same reasons as in $RQ_4$ and $RQ_5$.

The heatmap in Figure 15 shows the number of steps in function of their technical and opportunity lag. We observe that there are plenty of steps with a similar technical lag and opportunity lag (they are found on or close to the diagonal). At the same time many steps have either a high technical lag and low opportunity lag (shown on the bottom right of the heatmap), or vice versa (shown on the top left). The *weak* Pearson correlation ($r = 0.39$) and *moderate* Spearman correlation ($\rho = 0.49$) suggest that technical lag and opportunity lag are complementary metrics to quantify outdatedness of workflows w.r.t. the Actions used by them.

Still excluding the Actions from the actions/ provider, we computed the monthly evolution of the distribution of opportunity lag for all outdated steps in all considered snapshots. Figure 16 shows this evolution, revealing a continuous growth until June 2022, with a median value going from 0.5 months at the beginning of the observation to 9.3 months in May 2022.

We observe a sudden decrease in June 2022, indicated by a vertical dashed line. This decrease is the consequence of the new major releases of various Actions provided by docker/ that caused thousands of steps to become outdated (see $RQ_4$), inducing an opportunity lag of a few weeks for them in June 2022. From July onwards, the opportunity lag started to increase again, with a median

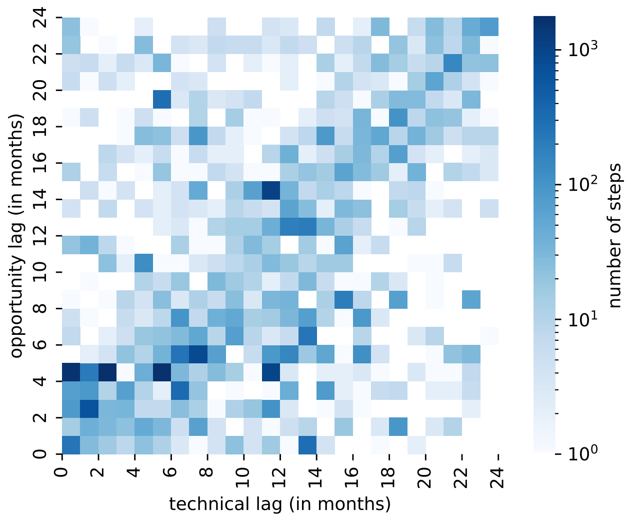

Figure 15: Technical lag versus opportunity lag for outdated steps in the latest snapshot, excluding steps using Actions from the actions/ provider.

and mean value respectively reaching 4.9 and 9 months at the end of the observation period. Phrased differently, this means that the maintainers of half of the outdated steps had more than 4.9 months of opportunity to update the versions of the Actions they used, but did not seize this opportunity. This duration even reaches more than 13.6 months for a quarter of the outdated steps.

If is difficult to know whether maintainers did not update these Actions on purpose. On the one hand, continuing to use outdated Actions incurs a higher risk of having bugs and security issues in them, but on the other hand updating to a more recent release might lead to backward incompatible changes or incompatibilities. Moreover, nothing indicates that the maintainers are actually aware of the outdatedness of their workflows, nor of the risks they may be facing as a consequence of this.

**Summary:** As a complementary metric to *technical lag*, the *opportunity lag* measures how long an outdated version of a reusable component has been used, while a newer version could have been used instead. The opportunity lag of outdated Actions used in workflow steps tends to increase over time. On average, maintainers of outdated steps have had the opportunity to update them for 9 months, but have not done so.

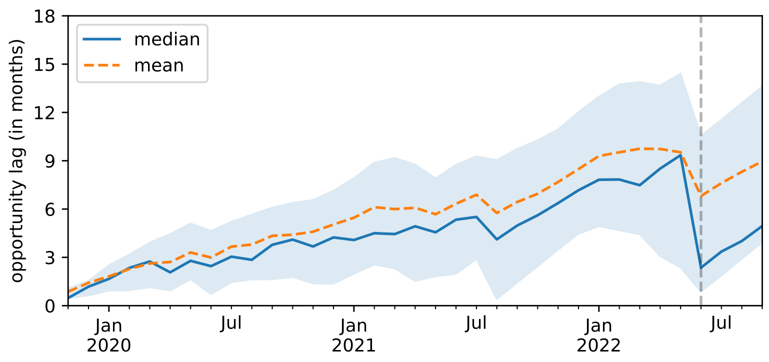

Figure 16: Evolution of the distribution of opportunity lag, excluding outdated steps using Actions from the actions/ provider. The shaded area corresponds to the interval between the 25th and 75th percentiles.

## 6. Discussion

### *6.1. The GitHub Actions software ecosystem*

Software component ecosystems are composed of large numbers of reusable interconnected software components that are maintained by large and often geographically distributed contributor communities. Examples of such ecosystems are package dependency networks of reusable software libraries for different programming languages (e.g., npm for JavaScript, Maven for Java, PyPI for Python, Cargo for Rust). Software applications can reuse these libraries by specifying *dependencies* towards specific (version ranges of) *releases* of those libraries. Despite their extreme usefulness for the software development community in the large, software library ecosystems have been shown to face numerous challenges related to dependency management [8, 9, 10], security vulnerabilities [11, 12, 13], backward compatibility [14, 15, 16], outdated components [17, 18, 19], deprecated and obsolete components [20] and loss of core developers [21].

GHA can be regarded as an ecosystem that bears many similarities with software library ecosystems. Actions are widely available for reuse, are developed and distributed by a large community of providers, and are used in automation workflows that specify dependencies to specific versions (releases) of these Actions. By analogy with software library ecosystems, it is therefore interesting to study the evolutionary characteristics of the GHA ecosystem, and the challenges that come with its use.

In this article, we have been able to confirm that GHA exhibits similar evolutionary characteristics (*continuing growth* and *continuing change*) as software library ecosystems [7]. For instance, Figures 2 and 4 reveal its *continuing growth*, Figure 7 highlights its *continuing change*, and Figure 3 shows that Actions are widely reused in workflow steps, forming a large dependency network.

With regard to the challenges that come in ecosystems of reusable components, we analysed the outdatedness of workflows with respect to the latest or highest available release of the Actions they use. $RQ_4$ revealed that four out of five workflows and nearly two thirds of the steps are using an outdated Action release. We relied on the concept of technical lag to assess to which extent reusable Actions used in workflows are outdated. $RQ_5$ showed that half of the outdated workflow steps are using a version that is lagging behind the latest one for at least 7.3 months. In $RQ_6$ we also proposed and evaluated the opportunity lag as a complementary metric to quantify how long workflow steps have had the opportunity to update the Actions they use. We found that maintainers of outdated steps have had the opportunity to update them for 9 months, but have not done so. All these findings confirm that the GHA ecosystem is suffering from the outdatedness induced by reusable Actions on workflows, to an extent that is similar to major software library ecosystems [47, 17].

In future work, we aim to study other challenges that the GHA ecosystem is facing, inspired by the many available studies about technical challenges in software library ecosystems, such as issues related to dependency management, security vulnerabilities, backward compatibility, deprecated and obsolete components, and loss of core developers.

**Recommendation:** GHA constitutes a software component ecosystem in its own right, that is in need of further study of numerous challenges, beyond the workflow outdatedness that has been studied in the present article.

### *6.2. Semantic Versioning practices*

GitHub recommends developers and maintainers of Actions to number their releases using a three-component version number, and $RQ_3$ revealed that this recommendation is widely followed. More specifically, we observed that most of the workflow steps rely on the single-component version notation (e.g., `@v2`) to refer to Action releases. This suggests that workflow maintainers generally assume that Actions adhere to SemVer. Under this assumption, breaking changes are expected to be limited to major releases. Therefore, using a single-component notation allows workflow maintainers to benefit from critical fixes and security patches in Actions while still retaining compatibility. However, prior research on software library ecosystems [42, 14] has observed that there is no guarantee that reusable components follow the semantics of SemVer even if they adhere to its syntax. This implies that breaking changes can still manifest themselves in minor or patch releases. The same may be true for the GHA ecosystem and therefore deserves a more in-depth analysis.

Unlike the dependency version constraint mechanisms provided by library ecosystems that advocate the use of SemVer (e.g., npm or Cargo), GitHub does not provide any appropriate way to specify and resolve version ranges in step anchors (e.g., `^1.2.3` or `[1.2.3-2.0.0[`), to refer to specific ranges of Action releases. Instead, GitHub recommends Actions maintainers to create additional git tags for "*keeping major (v1) and minor (v1.1) tags current to the latest appropriate commit*". To some extent, this recommendation circumvents the need for a genuine SemVer-based version specification and resolution mechanism. However, since this recommendation requires version resolution to be emulated through git tags, it comes with multiple major flaws:

1. Action maintainers have to create and move these git tags each time a new Action release is published. Unless automated, this introduces an additional burden on the developers.
2. There is no guarantee that these tags are moved consistently. Checking whether a partial version specification (e.g., `v1`) matches the latest version (e.g., `v1.2.3`) requires to compare the commit hashes targeted by these two tags.
3. Moving git tags should be considered as an *anti-pattern*. The official git manual even considers re-tagging as "*an insane thing*".[18] To make things even worse, git does not store any historical data about tags in its history log, making it impossible *a posteriori*, to keep track of the commits targeted by a tag.

In summary, it is surprising that GitHub, the largest git-based social coding platform, recommends versioning practices that contradict the intended and expected use of git tags. We share the viewpoint of other software practitioners to avoid the use of tags to simulate SemVer for Action releases and to advocate a built-in version resolution mechanism supporting range notations for step anchors.[19] As has been previously observed in software library ecosystems [47, 11], such a combination of SemVer with an appropriate dependency constraint mechanism is likely to reduce the outdatedness of workflows as well as their exposure to security vulnerabilities in the Actions they use.

**Recommendation:** Action and workflow maintainers would benefit from a consistent SemVer-based versioning policy and an appropriate version resolution mechanism supporting range notations.

## *6.3. Security impact of outdated Actions*

Our empirical results highlighted that the majority of workflows make use of reusable Actions, that these Actions are continuously updated ($RQ_2$) and that most workflows exhibit technical lag with respect to the Actions they use ($RQ_5$ and $RQ_6$). It has been shown for software library ecosystems that relying on an outdated dependency incurs a higher risk of having security vulnerabilities,

[18]`https://git-scm.com/docs/git-tag#_on_re_tagging`

[19]See `https://github.com/actions/toolkit/issues/214` for an interesting discussion on these topics, notably involving one of the Action engineers at GitHub.

and library maintainers may not even be aware that there are exposed to a vulnerability [25, 23, 24].

The same holds for GHA workflows relying on outdated Actions. According to GitHub Security Lab, "*a compromised or malicious action could potentially disrupt automatic workflows of your repository*" and "*all options [to refer to an Action with an anchor] are a tradeoff between guaranteed supply chain integrity and auto-patching of vulnerabilities in dependencies*" [51]. Qualitative and quantitative analysis has confirmed that security concerns are among the major challenges for developers when automating workflows on GitHub [36, 37]. Several respondents to an interview about the use of CI/CD tools [32] expressed security concerns due to the ability to rely on third-party Actions in workflows:

- *"you need to implement a whole ecosystem of security constraints because you can potentially be running arbitrary third-party code in your data center, so you need to make sure that you'll lock down that environment to make sure that the environment itself is actually secure."*

- *"It's a major security concern because, from that automation you can basically run any code on it."*

- *"Most of the GitHub Actions are kind of community ran on open source repositories. That makes me very nervous of using them because someone could push something malicious to the Action plugin which gets picked up automatically by default and then is ran on my project silently in the background."*

Multiple cases of security issues with potentially disastrous consequences have been reported for GHA. Examples include manipulating pull requests to steal arbitrary secrets [52], injecting arbitrary code with workflow commands,[20] or bypassing code reviews to push unreviewed code [53]. 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."*

There are also several known examples of Action releases suffering from security issues for which fixes have been made available in newer releases. We refer the reader to the GitHub Advisory Database that started to support reusable Actions in August 2022. For instance, hashicorp/vault-action has a known security vulnerability for all versions prior to `2.2.0`.[21] All workflows relying on a vulnerable release of this Action and making use of multi-line secrets are exposed to the high security risk of having these secrets revealed in the output log. While a fix has been released in version `2.2.0` in May 2022, one out of three workflows

[20]https://packetstormsecurity.com/files/159794/GitHub-Widespread-Injection.html
[21]https://github.com/advisories/GHSA-4mgv-m5cm-f9h7

in the September 2022 snapshot of our dataset still uses a compromised version of this Action. Another example is azure/setup-kubectl whose versions `v2` and lower suffer from a vulnerability that is fixed in `v3`.[22] Although version `v3` of the Action was released in June 2022, 29 out of the 34 workflows using it in our dataset (ending in September 2022) are still exposed to the vulnerability because they have not been updated to use `v3`.

Dependency monitoring and security monitoring tools (such as Renovatebot and GitHub's Dependabot) that keep track of the Action versions used and that warn workflow maintainers when new versions of these Actions are released, can help to reduce the outdatedness and exposure of workflows to security vulnerabilities. Such tools have been proven to be effective in software library ecosystems, leading projects to update their dependencies 1.6x more frequently than projects that did not use them [39].

Despite the addition of support for GHA to Dependabot in June 2020, the addition of alerts for vulnerable Actions in August 2022, and the ability to propose automatic updates for these Actions since November 2022, only 5.0% of the repositories in our dataset have configured Dependabot, and only 3.1% use it for monitoring Actions in workflows.

Another potential source of compromised security lies in the fact that the majority of workflows are using SemVer practices to refer to reused Actions. Since these practices are based on using mutable git tags, malevolent actors having gained access to a repository hosting an Action could delete tags or move them to point to a compromised commit [54]. GitHub announced in its roadmap to address this issue by supporting immutable Actions. [23] In the meanwhile, GitHub recommends workflow developers to implement security hardening by using unique commit hashes to refer to releases of Actions since "*it is currently the only way to use an Action as an immutable release*". [24] As such, this mechanism currently constitutes "*the safest for stability and security*", as it "*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*". This use of immutable commit hashes does not exclude using automated dependency updates by Dependabot or Renovatebot. To do so, it suffices to add a comment after the commit hash to indicate which release the hash refers to.

Our findings highlight an urgent need to increase the awareness of workflow maintainers to limit the use of vulnerable outdated Actions, for exemple by resorting to automated security and dependency monitoring tools. The security risk induced by relying on reusable Actions is further increased by the fact that, in addition to allowing to use Actions in workflows, GHA also allows to reuse workflows within a workflow,[25] as well as to reuse Actions within *composite*

[22]`https://github.com/advisories/GHSA-p756-rfxh-x63h`
[23]`https://github.com/github/roadmap/issues/592`
[24]`https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions#using-third-party-actions`
[25]`https://docs.github.com/en/actions/using-workflows/reusing-workflows`

*Actions*.[26] These two additional reuse mechanisms imply that workflow maintainers not only need to monitor and manage the Actions they use *directly* in their workflows, but they also need to consider the Actions they use *transitively* through reusable workflows and composite Actions. To do this, maintainers are currently left on their own, since dependency monitoring tools currently only consider Actions that are directly used in workflows, while ignoring their transitive dependencies.

**Recommendation:** Awareness needs to be increased among workflow maintainers to use dependency and security monitoring tools to limit the use of insecure outdated Actions. These tools should be improved to take into account the transitive use of Actions by workflows.

## *6.4. Beyond the GitHub Actions ecosystem*

The exposure of GHA to the well-known issues that software library ecosystems are known to face is worrisome because these issues will not remain limited to the GHA ecosystem but may also affect other software ecosystems. This situation is depicted in Figure 17.

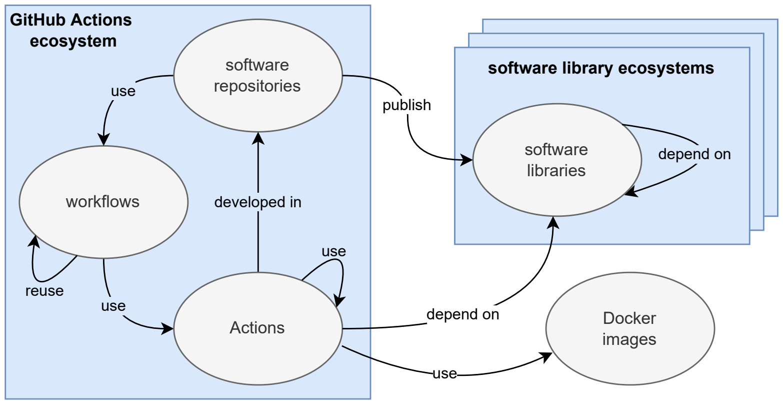

Figure 17: Interweaving of the GHA ecosystem and software library ecosystems.

Consider for example the case of some Action that is affected by a security vulnerability. This vulnerability may compromise all the workflows using the affected Action. Next, it may also compromise the software repositories in which these workflows are executed, leading an attacker to gain access to some of these repositories and to alter their code. By extension, attackers may create new compromised releases of the software projects being developed in the repositories, and publish them to some library registry. In turn, these libraries may

[26]https://docs.github.com/actions/creating-actions/creating-a-composite-action

affect all the dependent libraries that use them, and so on. For example, the dawidd6/action-download-artifact Action, used by hundreds of workflows, was found to expose workflows using it to code injection attacks.[27] While we are not aware of compromised library releases having been published to library registries because of this vulnerability, we would not be surprised to learn it did actually happen.

Conversely, the GHA ecosystem may be affected by issues coming from other ecosystems, as illustrated in Figure 17. Indeed, GHA allows developers to create Actions in three different ways, each of them increasing the risk and exposure to well-known issues:

1. A *composite Action* is defined through steps similar to those found in workflows. It can therefore not only execute commands, but also make use of other Actions, including other composite Actions. This opens the door to a potential contamination of the composite Action by the problems present in the Actions it uses.
2. A *Docker Action* executes the tasks described in a Docker image and the packages that compose it. Even if the execution within a Docker container is isolated from the execution of the workflow that uses it, the presence of bugs or vulnerabilities in the packages that compose the Docker image can compromise the Action that uses it. For instance, it has been shown that all Debian-based Docker images on DockerHub, even including official ones, contain several hundreds of packages with known vulnerabilities [45].
3. Finally, a *JavaScript Action* executes tasks implemented in JavaScript. Such an Action can make direct or indirect use of reusable libraries distributed through package managers such as npm, thereby increasing the attack surface of the Action. It has been shown that 15% of npm packages have a direct dependency on a package with a known vulnerability, and 36.5% are transitively exposed to a known vulnerability [12].

This strong interweaving between GHA and other ecosystems is not without practical consequences. Issues affecting either reusable libraries, Docker images, Actions or workflows may cross the GHA ecosystem boundaries and propagate to the software ecosystems it is interconnected with, leading to a substantially increased exposure to vulnerabilities and other socio-technical health issues.

Specifically for *composite* and *JavaScript Actions*, it has been shown that around 30% of these Actions have at least one high or critical security alert in their dependencies [55]: *“If your dependencies already are not up to date and thus have security issues in them, how can we expect your action to be secure?”* As a consequence, dependency monitoring tools should be adapted for workflows and Actions to take the cross-ecosystem transitive dependencies into account.

[27]https://www.legitsecurity.com/blog/github-actions-that-open-the-door-to-cicd-pipeline-attacks

**Recommendation:** The GHA ecosystem is strongly interwoven with other software component ecosystems. Issues affecting components in any of these ecosystems may cross the ecosystem boundaries and propagate to the connected software ecosystems. This calls for new cross-ecosystem studies to assess how and to which extent Actions affect, or are affected by, risks and issues related to component dependencies that cross ecosystem boundaries. Tools that monitor dependencies should be adapted to take into account these cross-ecosystem dependencies.

## 7. Threats to validity

We follow the structure recommended by Wohlin et al. [56] 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 workflows in GitHub repositories on the basis of the presence of a YAML file in their `.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 since there is little practical reason to keep workflows in that folder without actually using them.

We also implicitly assumed that all steps defined in jobs, all jobs defined in a workflow file, and all workflow files defined in a repository are equally important. However, workflows, jobs and steps may be used for various purposes [6] with different degrees of importance. It may be the case that some workflows were just created to "play around" with GHA, that some jobs tend to be executed more frequently (e.g., those making use of the `matrix` strategy), or that some steps are only conditionally executed (e.g., those making use of the `if` construct). As a consequence, we may have overestimated the importance of some Actions used in these steps, jobs and workflows.

The last threat to construct validity stems from how we identified releases for Actions. We explained in $RQ_2$ that we relied on the GitHub API for "releases" to detect when an Action has been updated. However, not all Actions are relying on the release system. We could only obtain releases for 79.9% of the Actions of our dataset, and we cannot claim that the list of releases we obtained for these Actions is complete. Nevertheless, we are confident about the representativeness of the identified releases since, as mentioned in $RQ_4$, we managed to map 92.6% of the versions used in steps to the releases of our dataset.

Threats to *internal validity* concern choices and factors internal to the study that could influence the observations we made. The findings of $RQ_3$ show that there is a discrepancy between how versions are specified in Actions and how they are specified in steps. This is a consequence of GitHub's recommendation to use SemVer-based anchors (i.e., `@v2` or `@v2.3`) to refer to the latest available release within the specified major or minor branch. These version "aliases" are usually made available through git tags. As explained in $RQ_4$, there is no historical data for git tags, making it impossible to identify the exact releases that are targeted by them. As a result, we mapped component-based versioning notations used in steps to the highest available release of the corresponding Actions using a prefix-based matching (e.g., `@v2` is mapped to `v2.3.4`), emulating the presence of appropriate git tags whenever an exact match cannot be found. This optimistic way of mapping versions assumes that GitHub's recommendation to create and move git tags is followed consistently by all Actions for which an exact match cannot be found. In cases where Actions do not define these additional tags, or do not correctly move them (e.g., `v2` points to `v2.3.4` instead of `v2.5.1`), this possibly leads to an underestimation of the outdatedness of the steps using them. As a consequence, the findings reported in $RQ_4$, $RQ_5$ and $RQ_6$ should be considered as a lower bound of the actual situation of outdatedness of workflows.

Another threat to internal validity relates to how we identified Actions used in workflows. While we considered all the Actions being used in workflow steps, we only considered the Actions being used *directly* and not those being used *transitively* through composite Actions or reusable workflows, as explained in Section 6. However, while composite Actions represent 11.7% of the Actions in our dataset, they are only used by 2.2% of the steps using an Action in the latest snapshot. Similarly, according to Decan et al. [6], only 0.9% of the workflows are reusing another workflow. Moreover, 84.4% of these reused workflows are located in the same repository as the calling workflow, implying that we did actually take most of them into account when identifying the Actions used in workflows.

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 limit the analysis to active GitHub repositories having at least 100 stars and 100 commits, in order to exclude abandoned, personal or experimental repositories [49]. This implies that we have no insight into how GHA is used in smaller or less active repositories, nor on the more specific Actions that might be used in these repositories (e.g., to publish personal GitHub pages or to compile LaTeX files).

## 8. Conclusion

Since its public release in November 2019, GHA has become the dominant CI/CD service on GitHub, only 18 months after its introduction. Its Marketplace of reusable Actions, and the amount of repositories and workflows using such Actions, has been growing rapidly ever since. Given the high number of

GitHub repositories reusing Actions, the GHA ecosystem bears many similarities to ecosystems of reusable software libraries (such as npm) that are known to suffer from various issues related to dependency management.

Based on a dataset of nearly one million workflows obtained from 22K+ repositories between November 2019 and September 2022, we empirically studied workflows in GitHub repositories, focusing on two research goals.

Goal $\mathbf{G_1}$ provided quantitative insights in the prevalence of reusable Actions, as well as their evolution over time in terms of update frequency and versioning practices being used. We quantitatively observed that it is common for workflows to use Actions, and that a limited number of Actions concentrate most of this reuse. We also found most Actions and workflows to be frequently updated. We investigated the versioning practices used by workflows to refer to the Actions they use, and observed a discrepancy between using a three-component and a single-component version notation to specify the releases of used Actions.

Goal $\mathbf{G_2}$ focused on the outdatedness of workflows in terms of the reusable Actions used by them. We quantified to which extent workflows are outdated, based on the technical lag metric, observing that most workflows are using an outdated release of an Action, lagging behind the latest available one for several months. We proposed the opportunity lag as a complementary way to measure how long a reused Action has been outdated. We used this metric to reveal that most workflows had the opportunity to update their used Actions for 9 months, but did not.

These findings indicate that most of the workflows that use Actions depend on outdated releases of those Actions, increasing their vulnerability risk. This calls for a more rigorous management of Action outdatedness, as well as for better policies and tooling to support it effectively. This is especially critical since workflows using outdated Actions can not only affect the repositories in which they are executed, but can also propagate beyond the boundaries of the GHA ecosystem.

As future work, we aim to study other dependency-related challenges that the GHA ecosystem is facing, and how and to which extent issues propagate to and from the GHA ecosystem to other software ecosystems. We also aim to get a better understanding on the changes occurring in the continuously evolving Actions and workflows, and on the implications of these changes. To do so, we plan to complement the quantitative evidence in this article with qualitative insights about the awareness of outdated Actions, the reasons of being outdated, and the perception of the security and other risks related to outdatedness.

## Acknowledgments

This research is supported by the Fonds de la Recherche Scientifique - FNRS under grant numbers MIS F.4515.23, O.0157.18F-RG43 and PDR T.0149.22, and by ARC-21/25 UMONS3 supported by the Ministère de la Communauté française – Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique.

## References

[1] J. M. Costa, M. Cataldo, 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] GitHub, Octoverse 2022: The state of open source software, `https://octoverse.github.com/2022/developer-community`, [Online; accessed 7 December 2022] (2022).

[3] M. Fowler, M. Foemmel, Continuous Integration, `https://martinfowler.com/articles/originalContinuousIntegration.html`, [Online; accessed 3 January 2022] (September 2000).

[4] M. Golzadeh, A. Decan, T. Mens, On the rise and fall of CI services in GitHub, in: International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2022. `doi:10.1109/SANER53432.2022.00084`.

[5] C. Chandrasekara, P. Herath, Hands-on GitHub Actions: Implement CI/CD with GitHub Action Workflows for Your Applications, Apress, 2021. `doi:10.1007/978-1-4842-6464-5`.

[6] A. Decan, T. Mens, P. R. Mazrae, M. Golzadeh, On the use of GitHub Actions in software development repositories, in: International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2022. `doi:10.1109/ICSME55016.2022.00029`.

[7] A. Decan, T. Mens, P. Grosjean, An empirical comparison of dependency network evolution in seven software packaging ecosystems, Empirical Software Engineering 24 (1) (2019) 381–416. `doi:10.1007/s10664-017-9589-y`.

[8] A. Decan, T. Mens, M. Claes, An empirical comparison of dependency issues in OSS packaging ecosystems, in: International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2017. `doi:10.1109/SANER.2017.7884604`.

[9] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, E. Shihab, Why do developers use trivial packages? An empirical case study on npm, in: Joint Meeting on Foundations of Software Engineering (FSE), 2017, pp. 385–395.

[10] C. Soto-Valero, N. Harrand, M. Monperrus, B. Baudry, A comprehensive study of bloated dependencies in the Maven ecosystem, Empirical Software Engineering 26 (3) (2021) 45. `doi:10.1007/s10664-020-09914-8`.

[11] A. Decan, T. Mens, E. Constantinou, On the impact of security vulnerabilities in the npm package dependency network, in: International Conference on Mining Software Repositories (MSR), 2018, pp. 181–191. `doi:10.1145/3196398.3196401`.

[12] A. Zerouali, T. Mens, A. Decan, C. De Roover, On the impact of security vulnerabilities in the npm and RubyGems dependency networks, Empirical Software Engineering 27 (5) (2022) 1–45. doi:10.1007/s10664-022-10154-1.

[13] M. Alfadel, D. E. Costa, E. Shihab, E. Shihab, Empirical analysis of security vulnerabilities in Python packages, in: International Conference on Software Analysis, Evolution and Reengineering (SANER), 2021. doi:10.1109/saner50967.2021.00048.

[14] A. Decan, T. Mens, What do package dependencies tell us about semantic versioning?, IEEE Transactions on Software Engineering 47 (6) (2019) 1226–1240. doi:10.1109/TSE.2019.2918315.

[15] A. Decan, T. Mens, A. Zerouali, C. De Roover, Back to the past–analysing backporting practices in package dependency networks, IEEE Transactions on Software Engineering 48 (10) (2022). doi:10.1109/TSE.2021.3112204.

[16] C. Bogart, C. Kästner, J. Herbsleb, F. Thung, When and how to make breaking changes: Policies and practices in 18 open source software ecosystems, ACM Transactions on Software Engineering and Methodology (TOSEM) 30 (4) (2021) 1–56.

[17] A. Decan, T. Mens, E. Constantinou, On the evolution of technical lag in the npm package dependency network, in: International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2018, pp. 404–414. doi:10.1109/ICSME.2018.00050.

[18] A. Zerouali, T. Mens, J. Gonzalez-Barahona, A. Decan, E. Constantinou, G. Robles, A formal framework for measuring technical lag in component repositories—and its application to npm, Journal of Software: Evolution and Process 31 (8) (2019). doi:10.1002/smr.2157.

[19] T. Lauinger, A. Chaabane, C. B. Wilson, Thou shalt not depend on me, Communications of the ACM 61 (6) (2018) 41–47. doi:10.1145/3190562.

[20] F. Cogo, G. Oliva, A. E. Hassan, Deprecation of packages and releases in software ecosystems: A case study on npm, IEEE Transactions on Software Engineering (2021). doi:10.1109/TSE.2021.3055123.

[21] G. Avelino, E. Constantinou, M. T. Valente, 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.

[22] M. Wessel, T. Mens, A. Decan, P. Rostami Mazrae, The github development workflow automation ecosystems, in: Software Ecosystems: Tooling and Analytics, Springer, 2023.

[23] R. G. Kula, D. M. German, A. Ouni, T. Ishio, K. Inoue, Do developers update their library dependencies?, Empirical Software Engineering 23 (1) (2018) 384–417. `doi:10.1007/s10664-017-9521-5`.

[24] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, X. Peng, Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem, in: International Conference on Software Engineering (ICSE), 2022, pp. 672–684.

[25] J. Cox, E. Bouwers, M. C. J. D. van Eekelen, J. Visser, Measuring dependency freshness in software systems, in: International Conference on Software Engineering (ICSE), IEEE, 2015, pp. 109–118.

[26] A06:2021 – vulnerable and outdated components, Open Worldwide Application Security Project (OWASP) (2021).
URL `https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/`

[27] O. Elazhary, C. Werner, Z. S. Li, D. Lowlind, N. A. Ernst, M.-A. Storey, Uncovering the benefits and challenges of continuous integration practices, IEEE Transactions on Software Engineering 48 (7) (2022) 2570 – 2583. `doi:10.1109/TSE.2021.3064953`.

[28] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, 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.

[29] M. Hilton, T. Tunnell, K. Huang, D. Marinov, 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.

[30] M. Shahin, M. A. Babar, L. Zhu, Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices, IEEE Access 5 (2017) 3909–3943.

[31] E. Soares, G. Sizilio, J. Santos, D. Alencar, U. Kulesza, The effects of continuous integration on software development: a systematic literature review, Empirical Software Engineering (2022).

[32] P. Rostami Mazrae, T. Mens, M. Golzadeh, A. Decan, On the usage, co-usage and migration of CI/CD tools: A qualitative analysis, Empirical Software Engineering 28 (2) (2023) 52. `doi:10.1007/s10664-022-10285-5`.

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

[34] T. Chen, Y. Zhang, S. Chen, T. Wang, Y. Wu, Let's supercharge the workflows: An empirical study of GitHub Actions, in: International Conference on Software Quality, Reliability and Security Companion (QRS-C), IEEE, 2021.

[35] P. Valenzuela-Toledo, A. Bergel, Evolution of GitHub Action workflows, in: International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2022.

[36] S. G. Saroar, M. Nayebi, Developers' perception of GitHub Actions: A survey analysis, in: International Conference on Evaluation and Assessment in Software Engineering (EASE), ACM, 2023. doi:10.1145/3593434.3593475.

[37] G. Benedetti, L. Verderame, A. Merlo, Automatic security assessment of GitHub Actions workflows, in: Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses, ACM, 2022, pp. 37–45. doi:10.1145/3560835.3564554.

[38] V. Kotovs, Forty years of software reuse, Sci. J. Riga Tech. Univ. 38 (38) (2009) 153–160.

[39] S. Mirhosseini, C. Parnin, Can automated pull requests encourage software developers to upgrade out-of-date dependencies?, in: International Conference on Automated Software Engineering (ASE), 2017, pp. 84–94. doi:10.1109/ASE.2017.8115621.

[40] M. P. Robillard, R. J. Walker, T. Zimmermann, Recommendation systems for software engineering, IEEE Software 27 (2010) 80–86.

[41] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, S. Panichella, The evolution of project inter-dependencies in a software ecosystem: The case of Apache, in: International Conference on Software Maintenance (ICSM), IEEE, 2013, pp. 280–289.

[42] S. Raemaekers, A. van Deursen, J. Visser, Semantic versioning and impact of breaking changes in the Maven repository, J. Syst. Softw. 129 (2017) 140–158.

[43] J. M. Gonzalez-Barahona, P. Sherwood, G. Robles, D. Izquierdo-Cortazar, Technical lag in software compilations: Measuring how outdated a software deployment is, in: Int'l Conference on Open Source Systems (OSS), Springer, 2017. doi:10.1007/978-3-319-57735-7_17.

[44] A. Zerouali, E. Constantinou, T. Mens, G. Robles, J. M. Gonzalez-Barahona, An empirical analysis of technical lag in npm package dependencies, in: International Conference on Software Reuse (ICSR), 2018. doi:10.1007/978-3-319-90421-4_6.

[45] A. Zerouali, T. Mens, A. Decan, J. M. Gonzalez-Barahona, G. Robles, A multi-dimensional analysis of technical lag in Debian-based Docker images, Empir. Softw. Eng. 26 (2021).

[46] J. M. Gonzalez-Barahona, Characterizing outdateness with technical lag: an exploratory study, International Conference on Software Engineering Workshops (2020). `doi:10.1145/3387940.3392202`.

[47] J. Stringer, A. Tahir, K. Blincoe, J. Dietrich, Technical lag of dependencies in major package managers, in: Asia-Pacific Software Engineering Conference (APSEC), 2020, pp. 228–237. `doi:10.1109/APSEC51365.2020.00031`.

[48] R. Gonzaga, Why you should pin your github actions by commit-hash, `https://blog.rafaelgss.dev/why-you-should-pin-actions-by-commit-hash` (June 2023).

[49] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, D. Damian, The promises and perils of mining GitHub, in: International Conference on Mining Software Repositories (MSR), ACM, 2014, pp. 92–101. `doi:10.1145/2597073.2597074`.

[50] O. Dabic, E. Aghajani, G. Bavota, Sampling projects in GitHub for MSR studies, in: International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 560–564.

[51] J. Lobacevski, Keeping your github actions and workflows secure. part 3: How to trust your building blocks, `https://securitylab.github.com/research/github-actions-building-blocks/` (August 5 2021).

[52] T. Katz, Stealing arbitrary GitHub Actions secrets, `https://blog.teddykatz.com/2021/03/17/github-actions-write-access.html` (March 17 2021).

[53] O. Gil, Bypassing required reviews using github actions, `https://medium.com/cider-sec/bypassing-required-reviews-using-github-actions-6e1b29135cc7` (October 12 2021).

[54] P. Elliott, Compromise by git tags, `https://www.scalefactory.com/blog/2021/02/18/compromise-by-git-tags/` (Feburary 18 2021).

[55] R. Bos, Github actions has security issues, XPRT Magazine 13 (2022) 37–39.
URL `https://xpirit.com/wp-content/uploads/2022/10/Xpirit_XPRT_magazine_13_final.pdf`

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