A Preliminary Study of GitHub Actions Dependencies

Hassan Onsori Delicheh[1], Alexandre Decan[1,2] and Tom Mens[1]

[1] *University of Mons (UMONS), Mons, Belgium*
[2] *F.R.S.-FNRS Research Associate*

Abstract
GitHub Actions was introduced in 2019 as a software development workflow automation tool, allowing to automate a wide range of social and technical activities in GitHub repositories. The reusable components known as Actions are developed within GitHub repositories and can be distributed through the GitHub Marketplace. GitHub Actions forms an ecosystem because workflows can rely on reusable Actions, that themselves may depend on other components such as NodeJS packages, Docker images, or other Actions. Just as packages in software library ecosystems have been shown to suffer from a multitude of maintainability issues due to their complex dependency networks, we posit that the same is true for Actions. Therefore, this paper presents preliminary insights in the dependencies of Actions. Based on a dataset of 2,817 Actions, we report on the characteristics of these Actions and we explore to which extent they are developed using JavaScript, Docker or as composite Actions, and to which extent they depend on other components. We show that most Actions are developed using JavaScript, and that composite Actions are gradually replacing Docker Actions. We also show that Actions have many dependencies, especially towards JavaScript packages, resulting in a large number of deeply nested transitive dependencies. This justifies the need for further maintainability studies of the GitHub Actions ecosystem.

Keywords
dependency management, GitHub Actions, software ecosystem, collaborative software development, workflow automation

1. Introduction

Open-source software (OSS) has gained significant popularity and typically forms a substantial portion of a modern software application stack, ranging from 70% to 90% of its code base [1]. Software development has evolved into a persistent, extensively distributed and collaborative endeavour [2]. During collaborative development, a multitude of tasks must be executed, including coding, debugging, testing, quality and security analysis, packaging and releasing software distributions, and so forth. These tasks require the use of version control systems, software distribution managers, bug and issue trackers, and quality, vulnerability and dependency analysers. Many of these tools are integrated into, or accessible through, social coding platforms [3]. The largest such platform to date is GitHub, hosting millions of software repositories and having served over 94 million users in 2022 [4].

SATToSE'23: Seminar on Advanced Techniques & Tools for Software Evolution, June 12–14, 2023, Salerno, Italy
hassan.onsoridelicheh@umons.ac.be (H. O. Delicheh);
alexandre.decan@umons.ac.be (A. Decan);
tom.mens@umons.ac.be (T. Mens)
0000-0002-5824-5823 (A. Decan);
0000-0003-3636-5020 (T. Mens)

To speed up the pace of development while maintaining high-quality software releases, continuous integration, deployment, and delivery (CI/CD) was introduced to automate a plethora of repetitive development-related tasks [5]. With the introduction of GitHub Actions in 2019, GitHub has integrated CI/CD support within GitHub repositories, providing direct access to a wide range of services, including automated building, testing, quality analysis, code review, communication, licence verification, and monitoring dependencies and security vulnerabilities. Only 18 months after its public release, GitHub Actions has become the dominant CI/CD service on GitHub [6].

GitHub Actions facilitates the creation of workflows by providing reusable Actions, that are distributed through the GitHub Marketplace.[1] In January 2023, already over 17K Actions were available for reuse. There is a significant amount of community involvement in the development of these Actions, which entails a considerable risk of malicious actors that intentionally develop new or modify existing Actions to compromise software development repositories.

Actions can be developed in three different ways: using JavaScript code, using Docker containers, or through the mechanism of *composite Actions*. Addi-

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

tionally, Actions may depend on other Actions or on third-party software components such as npm packages or Docker images. Relying on such third-party components may lead to maintainability issues such as incompatibilities, security vulnerabilities, and the reliance on outdated or abandoned components [7]. For instance, a study focusing on the security vulnerabilities of *composite Actions* and *JavaScript Actions* revealed that around 30% of these Actions had at least one high or critical security alert in their dependencies [8]. This makes it crucial to analyse the dependencies of Actions. This justifies the current article, which provides a preliminary study of the dependency network of Actions that are developed through GitHub repositories and published on the GitHub Marketplace. Such an investigation serves as an initial and fundamental stepping stone for researchers seeking to comprehend the potential impact of maintainability issues in the GitHub Actions ecosystem.

We carry out a quantitative empirical analysis with two main research goals. Goal **G1** aims to characterise Actions distributed through the GitHub Marketplace, quantifying the different approaches for developing Actions and their evolution over time. This goal is subdivided into two research questions that are centered around a quantitative analysis of the characteristics of 2,817 Actions on the GitHub Marketplace, as well as their 25,975 corresponding releases on the GitHub repositories in which they are developed.

$RQ_{1.1}$ What are the characteristics of Actions distributed through the GitHub Marketplace?

$RQ_{1.2}$ How do different types of Actions evolve?

Goal **G2** focuses on the dependency characteristics for the three types of Actions, through the following research questions:

$RQ_{2.1}$ What are the dependency characteristics of composite Actions?

$RQ_{2.2}$ What are the dependency characteristics of JavaScript Actions?

$RQ_{2.3}$ What are the dependency characteristics of Docker Actions?

2. Related work

On the usage and adoption of GitHub Actions. Previous research on GitHub Actions has primarily focused on the usage and adoption of GitHub Actions, with limited investigation into their development. Several empirical studies have evaluated the adoption and usage details of Actions, typically by analysing data from GitHub repositories that use and adopt them for specific automation tasks. For example, Golzadeh et al. [6] analysed 91,810 GitHub repositories to study how the CI/CD landscape has changed since the introduction of GitHub Actions. They found that the adoption of Actions is associated with a decline of other CI/CD tools such as Travis, CircleCI, and Azure. Rostami et al. [9] conducted a qualitative study aimed at comprehending the underlying factors driving the transformations in the CI/CD landscape. In-depth interviews with 22 experienced software practitioners provided insights into their usage, co-usage, and migration patterns of 31 distinct CI/CD tools. Based on this qualitative analysis, they identified a discernible trend of migrating towards GitHub Actions, and they also pinpointed the primary drivers behind this trend.

Kinsman et al. [10] examined 3,190 repositories to investigate changes in various development activity indicators following the adoption of Github Actions. Chen et al. [11] investigated the effects of GitHub Actions on GitHub projects. Valenzuela-Toledo and Bergel [12] conducted a study on the usage and maintenance practices of GitHub Actions workflows in popular GitHub repositories and identified various types of workflow modifications. Benedetti et al. [13] proposed a security assessment methodology to investigate the impact of security issues on GitHub Actions workflows and the software supply chain. Decan et al. [14] studied the usage of GitHub Actions in GitHub repositories. They characterised the repositories and their workflows by analysing job types, steps and used Actions. They observed that almost all workflows use Actions, which could potentially pose a problem as any issues with these Actions, including bugs, security vulnerabilities, or outdated components, could negatively impact the workflows that incorporated them.

Saroar and Nayebi [15] investigated the motivations, decision criteria, and challenges associated with creating, publishing, and using GitHub Actions. They discovered that, when presented with comparable options, developers tend to favour Actions created by verified individuals and having a higher number of stars. Furthermore, they noticed that users frequently switch to alternative Actions in response to issues such as bugs and insufficient documentation. Additionally, they found that configuring and debugging workflow files is one of the most common difficulties encountered by users of GitHub Actions. A recent study on security checks for Actions in the Marketplace also discovered that when Actions are forked and *Dependabot* [2] is enabled on the forked repositories, particularly for

[2] https://docs.github.com/en/code-security/dependabot/dependabot-security-updates/about-dependabot-security-updates

composite and JavaScript Actions, approximately 30% of these Actions contain at least one high or critical security alert in their dependencies [8]. To the best of our knowledge, our current quantitative study is the first to focus on how Actions are developed, differentiating between different types of Actions and examining the characteristics, evolution and dependencies in the GitHub Actions ecosystem.

On the dependency management of software library ecosystems. Since this paper aims to investigate dependencies in the GitHub Actions ecosystem, it is worthwhile to explore prior studies that have explored dependencies in other software ecosystems.

It is a widely adopted practice among software developers to depend on reusable software components to benefit from pre-existing code, rather than creating everything from scratch [16]. To enhance this practice of reuse, package managers and registries of reusable libraries have been introduced for the predominant programming languages, such as npm for JavaScript, PyPI for Python, and Maven for Java. While software component reuse provides several benefits [17], it also comes with maintainability issues such as dependency management [18, 19, 20], security vulnerabilities [21, 22, 23, 24, 25, 26], compatibility issues [27, 28, 29], outdated components [30, 31], deprecated and obsolete components [32]. Mirhosseini and Parnin [33] studied the incentives of software developers to update their project dependencies. The practice of maintaining outdated dependencies increases the risk of encountering bugs and security vulnerabilities. Cox et al. [34] analysed Java projects and found that projects relying on outdated dependencies were four times more likely to experience security issues and compatibility problems.

3. About GitHub Actions

GitHub Actions is a CI/CD tool integrated into GitHub to allow maintainers of GitHub repositories to automate a wide range of tasks. Following the "configuration as code" paradigm, workflows are specified as YAML files (stored in the .github/workflows folder of the repository). A workflow reacts to one or more *events* (e.g., a pull request is submitted, commits are pushed, or an issue is opened) and executes one or more *jobs*. Jobs are defined in terms of the *steps* that will be executed when the workflow is triggered. A step can either define the shell commands that need to be executed (using the `run:` syntax) or can refer (using the `uses:` syntax) to a reusable component, called an Action, to carry out its task.

The GitHub Actions Marketplace is a centralised platform that enables users to browse, discover, and share reusable Actions within the GitHub ecosystem. It serves as a valuable resource for GitHub community to enhance their productivity and streamline their workflows through the use of reusable Action. It includes features for filtering, sorting, and searching Actions based on various criteria, such as category and popularity based on stars. In the GitHub Actions Marketplace, Actions can be assigned a primary and a secondary category. The Actions that are published in the GitHub Marketplace typically include information such as the name of the Action, its contributors, its primary and secondary category, a brief description of its functionality, information about the developer or organization that created the Action, the corresponding GitHub repository, the number of stars, the number of open issues, the number of pull requests, and the latest 10 releases,.

Actions can be developed in any public GitHub repository and shared on the GitHub Marketplace. To enable an Action to be reused, one has to create a YAML file named action.y(a)ml at the root of the repository. This file details the metadata of the Action, such as its name, its set of parameters and its type. An Action can be developed in three different ways (i.e., it can be one of three types):

JavaScript Actions enable the execution of JavaScript code within a Node.js runtime environment. They are used for tasks requiring complex logic or interactions with the GitHub API or other external services. For example, a JavaScript Action could be created to automate the process of creating a new issue in a project management tool when a new pull request is opened. The use of JavaScript also unlocks the potential to rely on a huge number of JavaScript libraries and packages distributed through package managers such as npm.

Docker Actions define tasks that are executed in a Docker container. This allows for greater flexibility and portability in workflow execution, as the environment can be customised to suit the needs of the workflow. A *Docker Action* can be developed using a *Dockerfile*, which defines the base image of the container, and the various commands that will be executed on top of this image. These base images can be found in container registries, such as Docker Hub, GitHub Container Registry, Google Container Registry, Microsoft Container Registry, Red Hat Quay and Harbor.

Composite Actions allow to combine multiple workflow steps within one Action, similar to how jobs are defined in workflow files. They allow developers to simplify complex workflows and reduce

code duplication by defining such behaviour directly in a YAML file.

4. Data extraction process

To study the three types of Actions and their dependencies, it is required to obtain a large dataset of Actions and their respective releases from GitHub repositories where the Actions were developed. In order to acquire such a dataset, we retrieved Actions that are distributed through the GitHub Marketplace, which is the central location for discovering reusable Actions. At the time of writing, there was no API to obtain all Actions distributed on the Marketplace. Therefore, we gathered the Actions contained in each category listed on the Marketplace. In order to exclude Actions that might have been created only for personal or experimental purposes [35] we only considered Actions with at least 10 stars.

Through this process, we extracted 2,817 distinct Actions and their associated metadata including their primary and secondary categories, the name of the repository in which the Action is developed, its number of stars, issues, pull requests, and so on. Then, for each Action, we extracted its complete list of releases. To do so, we relied on the GitHub API for Releases, and obtained the 25,975 releases that were created between November 2019 (the official release date of GitHub Actions) and January 2023 (the data extraction date).

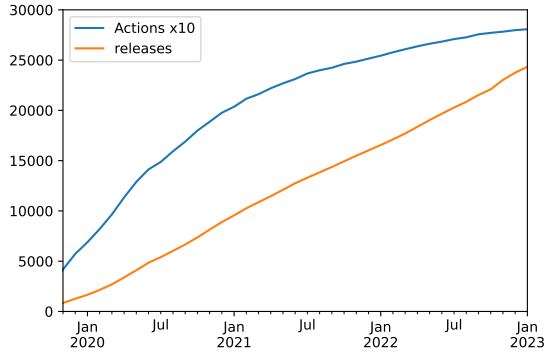

Figure 1: Evolution of the number of Actions on the Marketplace and their number of releases (in their GitHub repositories).

Figure 1 shows the evolution of the number of Actions that we extracted from the Marketplace (blue line, scaled by a factor of 10 for ease of comparison) and their number of releases (orange line). We observe a continuous growth in the number of Actions and releases.

5. Goal 1: Characterising Actions and their evolution

Goal $\mathbf{G_1}$ aims to characterise the Actions distributed through the GitHub Marketplace and quantify the three types of Actions and how they have evolved over time. Through a quantitative empirical analysis of the 2,817 considered Actions and their 25,975 releases extracted from the corresponding GitHub repositories, we answer the two following research questions.

$RQ_{1.1}$: What are the characteristics of Actions distributed through the GitHub Marketplace?

Since each type of Action uses a distinct approach for defining their dependencies, this research question provides a comprehensive overview of the Actions in our dataset. Investigating the characteristics of Actions distributed through the GitHub Marketplace can provide valuable insights into the current landscape of Actions.

Table 1 reports on the median and mean values for several characteristics such as the number of stars, the number of releases, the age (in months) and the type of Actions. The first line ("all categories") includes all Actions in our dataset. The next five lines list these characteristics for the top five categories comprising at least 5% of the Actions as either primary or secondary category. The last line corresponds to the Actions belonging to the remaining 18 categories. Notice that since an Action can belong to more than one category, the total number of Actions exceeds 2,817.

The categories of *continuous integration*, *utilities*, *deployment*, *publishing* and *code quality* are the most commonly used for publishing Actions on the GitHub Marketplace. In terms of popularity, the Actions in the *utilities* category had the highest average (157.5) and median (30) number of stars. We also found that the median and mean number of releases for Actions were 5 and 9.2, respectively. This indicates that Actions are still maintained and updated after their initial release. Regarding the type of Actions, *JavaScript Actions* and *composite Actions* are the most and least prevalent types of Actions, respectively, across the majority of categories. In the *publishing* and *code quality* categories, *Docker Actions* have a higher proportion than *JavaScript Actions*.

Table 1
Characteristics of Actions in the top 5 most popular Action categories on the GitHub Marketplace.

category	Actions #	Actions %	# stars median	# stars mean	# releases median	# releases mean	age (months) median	age (months) mean	% Action type composite	% Action type Docker	% Action type JS
all categories	2817	100.0	27	121.1	5	9.2	34.5	31.9	11.1	36.2	52.7
continuous integration	766	17.7	28	116.9	5	9.5	34.6	32.0	12.1	35.5	52.4
utilities	645	14.9	30	157.5	5	10.6	34.8	32.8	9.2	28.8	62.0
deployment	455	10.5	28	119.9	4	7.8	36	33.3	6.8	45.3	47.9
publishing	311	7.2	27	126.6	5	10.2	35.5	32.8	9.3	46.0	44.7
code quality	264	6.1	28	129.1	5	10.5	34.3	31.2	18.9	45.1	36.0
remaining 18 categories	1885	43.6	27	109,9	5	9.0	32.7	30.4	11.9	35.1	53.0

Overall, our observations did not reveal any significant differences between the various categories of Actions. Approximately half of all analysed Actions had at least 27 stars, 5 releases and were developed for a duration of 3 years. A majority of the Actions were developed as *JavaScript Actions*, accounting for 52.7% of the total, followed by *Docker Actions*, which constituted 36.2% of the Actions and composite Actions, which made up the remaining 11.1%. However, these proportions differed across the distinct categories of Actions.

$RQ_{1.2}$: How do different types of Actions evolve?

By analysing the YAML files of the latest releases of Actions in our dataset, we observed that 11.1%, 36.2%, and 52.7% of the considered Actions were released as *composite*, *Docker*, and *JavaScript* Actions, respectively. We also quantified the median and mean number of releases for the three types of Actions and observed that, although *composite Actions* is a new type of Action introduced in August 2020[3], they had more releases with a median of 5 and a mean of 9.4, as compared to *Docker Actions* with a median of 4 and a mean of 7.9.

Action developers may decide to change the type of Action over its lifetime. Therefore, investigating the migrations among the three different types of Actions and their evolution over the course of the observation period supplements the investigation of characteristics of Actions. Action developers may change the type of Action they are developing for a variety of reasons. The choice of Action type may depend on factors such as the required functionality, performance considerations, maintenance requirements and integration with other tools or services. Different types of Actions may be better suited for different use cases. For example, a *Docker Action* may be useful for running a containerised application, while a *JavaScript Action* might be more appropriate for customising the behavior of a workflow. Different types of Actions may also have varying performance characteristics. For example, a *JavaScript Action* may be faster than a *Docker Action* in some scenarios. Maintenance may also be a factor that can influence the choice of Action type. Some Action types may be more complex and time-consuming to maintain than others, which may prompt developers to switch to another type of Action that is easier to maintain. Finally, integration with other tools or services may also be a reason for changing the Action type. If an Action needs to work with another tool or service that works better with a different type of Action, developers may choose to switch to a more appropriate Action type.

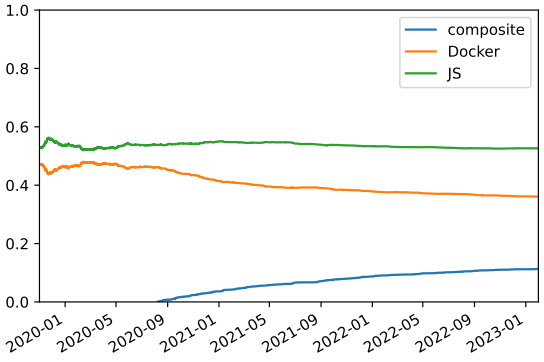

Figure 2: Evolution of different types of Actions over time.

Figure 2 shows the evolution over time of the proportion of Actions for each of the three types. We observe that the proportion of *JavaScript Actions* remains quite stable through time. On the other hand, the introduction of *composite Actions* in August 2020 led to a gradual decline in the proportion of *Docker Actions* in favour of *composite Actions*. The consistent increase in the proportion of *compos-*

[3]https://github.blog/changelog/2020-08-07-github-actions-composite-run-steps/

ite Actions is possibly attributed to their reusability, customisability and extensibility. They enable the definition of a set of steps that can be reused across multiple workflows and repositories, facilitating code reuse and reducing duplication. Moreover, regarding the ability to combine shell commands and using Actions, they can be customised to specific workflows or repositories and extended with additional functionality or steps.

Table 2 presents the number of Actions that migrated away from and towards an Action type as well as their relative proportion. We observe that *composite Actions* represent the majority of the targets of a migration, regarding the source type, *composite Actions* attracted 45.3% (73 out of 161) of all migrations away from *Docker Actions* and 57.4% (31 out of 54) of all migrations away from *JavaScript Actions*. Additionally, we note that 15.8% of the *Docker Actions* migrated to either *composite* or *JavaScript Actions*.

Table 2
Number and relative proportion of Actions that migrated away from and towards a specific type of Action.

		TO			Actions	
		composite	Docker	JS	#	%
FROM	composite	-	12	14	26	8.3
	Docker	73	-	88	161	15.8
	JS	31	23	-	54	3.6
Actions	#	104	35	102		
	%	33.3	3.4	6.9		

6. Goal 2: Analysing dependency characteristics of Actions

Goal $\mathbf{G_2}$ aims to characterise the dependency of the three Action types by analysing the GitHub repositories in which these Actions are being developed. Given that different Action types have different ways to depend on reusable components, we divide this goal into three research questions, one per Action type.

$RQ_{2.1}$ What are the dependency characteristics of composite Actions?

Composite Actions represent 11.1% of all Actions (i.e., 312 out of 2,817). Their YAML files define a set of steps that combine shell commands and reusable Actions. We extracted all steps from the *composite Actions*, distinguishing between shell commands (through the `run:` keyword) and used Actions (through the `uses:` keyword). The used Actions were considered as the dependencies of the *composite Actions*.

We found that 73% of the *composite Actions* (227 out of 312) are exclusively composed of 1 to 12 steps executing shell commands. The remaining 27% (i.e., 85 out of 312) combine both shell commands and used Actions. For those *composite Actions* that use (i.e., depend on) other Actions, we identified these dependencies and their type. If the dependencies were again *composite Actions*, we iteratively extracted all steps from their YAML file to obtain a complete list of indirect dependencies of *composite Actions*.

Table 3 reports on the dependency depth for *composite Actions*, the proportion of *composite Actions* at each depth and the types of Actions that *composite Actions* depend on. The first line, for instance, indicates that there are 85 *composite Actions* (accounting for 27% of the *composite Actions*) that have dependencies at depth 1 (i.e., the first level of dependency nesting). There are 177 dependencies required by these 85 Actions, comprising 23 *composite Actions*, 5 *Docker Actions*, and 149 *JavaScript Actions*. In terms of indirect dependencies, we discovered that 7.4% of *composite Actions* demonstrated such dependencies, relying indirectly on only 25 *composite Actions*.

Table 3
Dependency characteristics of *composite Actions*.

	composite Actions		type of dependency		
depth	#	%	composite	Docker	JS
1	85	27	23	5	149
2	23	7.4	2	0	10
3	2	0.06	0	0	0

Regarding the development approach for *composite Actions*, our analysis indicates that they are primarily developed to streamline complicated workflows and reduce code duplication, rather than heavily relying on other Actions available within the GitHub ecosystem.

$RQ_{2.2}$ What are the dependency characteristics of JavaScript Actions?

JavaScript Actions are written in JavaScript and may depend on npm packages. Developers usually define these included dependencies in the *package.json* file, the usual manifest file for JavaScript projects. The *dependencies* part of this manifest lists the name and version number of the required packages.

JavaScript packages and projects are known to be exposed to security vulnerabilities and other issues

coming from their dependencies [25, 21]. Therefore, we aim to quantify to which extent *JavaScript Actions* are relying on npm packages, as a preliminary step towards quantifying their exposure to security vulnerabilities and the impact of these vulnerabilities on the GitHub Actions ecosystem.

For each *JavaScript Action*, we extracted its list of dependencies from the *package.json* file of its latest release. This allowed us to get a list of all direct dependencies these Actions have. To obtain their indirect dependencies, we applied the `npm-remote-ls` [4] command-line tool on each of its direct dependencies. This tool produces a list of all the packages that are required, taking into account the dependency constraints, and also including those that are transitively required. For 1,364 out of 1,485 *JavaScript Actions* in our dataset, we collected a total of 90,370 (direct and indirect) dependencies in this way, accounting for 4,309 distinct required packages. The remaining 121 *JavaScript Actions* do not have a *package.json* file or the corresponding package did not available on npm registry. Therefore, they are not considered in the following analysis.

Table 4
Dependency characteristics of *JavaScript Actions*.

	JS Actions		# dependencies		
depth	#	%	median	mean	max
1	1364	100.0	4	4.3	27
2	1351	99.0	9	12.9	179
3	1347	98.7	12	18.4	266
4	1034	75.8	8	16.6	313
5	975	71.5	4	10.8	291
6	876	64.2	3	7.3	208
7	404	29.6	7	10.2	201
8	319	23.4	6	8.0	117
9	274	20.1	3	4.0	64
10	41	3.0	3	4.4	25
11	20	1.5	2	3.3	11
12	12	0.9	1	2.2	8
13	4	0.3	1	1.5	3

Table 4 reports on the depth and proportion of *JavaScript Actions* that depend on npm packages in the corresponding depth, along with the median, mean, and maximum number of dependencies associated with the depth of the dependency trees. For instance, the fifth row shows that among the *JavaScript Actions* in our dataset, 975 of them (representing 71.5% of the *JavaScript Actions*) have indirect dependencies at a depth of 5, with a median and mean value of 4 and 10.8 npm packages, respectively.

Our analysis also shows that the median and mean number of direct dependencies for *JavaScript Actions* were 4 and 4.3, respectively. Furthermore, we discovered that 99% of the *JavaScript Actions* contained indirect dependencies, a significant proportion of these dependencies being deeply nested. Specifically, we observed that up to 64% of the *JavaScript Actions* included indirect dependencies at a depth of 6. The *JavaScript Actions* are considerably exposed to potential issues associated with their dependencies, and since these dependencies may be deeply nested, it could become increasingly challenging to identify and resolve such issues. This situation is consistent with the observations made for JavaScript projects [25, 21].

In addition, among the npm packages specified as dependencies for *JavaScript Actions*, *@actions/core* was the most frequent npm depencency and 97% of *JavaScript Actions* in our dataset were dependent on that. The top five most frequently occurring npm packages are *@actions/core*, *@actions/github*, *@actions/exec*, *@actions/tool-cache* and *@actions/io*. The *@actions/* namespace corresponds to packages distributed by GitHub to ease the development and maintenance of *JavaScript Actions*, explaining why these packages are frequently found as dependencies for *JavaScript Actions*. For comparison, the most required packages that do not belong to this namespace are *semver* (used by 9.7% of *JavaScript Actions*), *axios* (used by 7.8% of *JavaScript Actions*) and *node-fetch* (used by 5.6% of *JavaScript Actions*).

$RQ_{2.3}$ What are the dependency characteristics of Docker Actions?

Docker Actions realise their tasks through the execution of a Docker container. To define a *Docker Action*, one has to specify, in the *action.y(a)ml* file at the root of the repository, either the URL to a Docker image (e.g., from Docker Hub) or the filepath to a *Dockerfile*. A *Dockerfile* is a file specifying the configuration of the container to be used for the Action. Among other, this file specifies the base image that should be used to create the Docker container, as well as the various commands that should be executed by this container.

Docker images are usually based on Linux distributions and contain several pre-installed packages that are required during the execution of the corresponding Docker containers. As such, we consider these images and their packages as dependencies for the *Docker Actions* that use them. However, getting the complete list of packages that are required that

[4] https://www.npmjs.com/package/npm-remote-ls

way by *Docker Actions* is not straightforward. As we will see, *Docker Actions* use a large number of images and these images are based on several Linux distributions, each of them having its own package manager and requiring a different methodology to identify the packages that are part of the image. This research question therefore focuses on the images used in *Docker Actions*, keeping for future work the identification of the packages that are actually used within the Docker containers.

We found that the YAML files of 156 out of 1,020 (i.e., 15.3%) of the *Docker Actions* specify the URL of the Docker image directly. The remaining 864 *Docker Actions* refer to a *Dockerfile* instead. We extracted this *Dockerfile* for 842 out of the 864 cases. For the remaining 22 Actions, we could not find the corresponding *Dockerfile* on their repositories.

For each *Dockerfile*, we extracted the URL of the base image, i.e., the value of the `FROM` field of the *Dockerfile*. We parsed this field and extracted the name of the image and the registry where the image can be found. We did the same for the Actions that directly specify the URL of the Docker image in their YAML file.

Table 5 lists the Docker registries that are the most frequently used by *Dockerfile*s and *action.y(a)ml* files. We observe that Docker Hub is the most frequently used Docker registry. It is used in 95% of the extracted *Dockerfile*s and in 57.7% of the Action YAML files. This is expected, since Docker Hub is the default registry that is used for retrieving Docker images when no registry is explicitly specified. Moreover, Docker Hub is one of the largest and most active Docker registries. Additionally, during our investigation, we discovered that GitHub Container Registry (ghcr.io) ranks as the second most commonly used Docker registry, used in 36.5% of the Action YAML files. Other registries being reported are the Microsoft Container Registry (MCR), docker.io and Google Container Registry.

Table 5
Docker image registries used by *Docker Actions*.

	Dockerfile		*action.y(a)ml*	
container registry	#	%	#	%
Docker Hub	800	95.0	90	57.7
GitHub Container Registry	16	1.9	57	36.5
Microsoft Container Registry	15	1.8	0	0.0
docker.io	3	0.3	3	1.9
Google Container Registry	3	0.3	1	0.6
other registries	5	0.7	5	3.3

We also quantified the base images specified in the *Dockerfile*. Table 6 reports the top ten most commonly used images by *Docker Actions* along with the proportion of *Docker Actions* that used them. The numbers suggest that the predominant referenced images are mostly associated with both programming languages (e.g., python, node, golang, etc.) and operating systems (e.g., alpine, ubuntu, etc.).

Table 6
Most frequent images used by *Docker Actions*.

	Docker Actions	
docker image	#	%
alpine	157	18.7
pyhton	140	16.7
node	88	10.5
golang	66	7.8
ubuntu	32	3.8
ruby	22	2.6
docker	22	2.6
debian	21	2.5
openjdk	9	1.1
php	8	1.0
all other types of images combined	277	32.7

Previous research [36, 37, 38] has primarily emphasised the security of system packages inside Docker images, disregarding third-party packages. Considering the growing prevalence of these packages, it is crucial to examine maintainability issues of such third-party packages and the impact of their use in the GitHub Actions ecosystems. The current analysis represents an initial step in studying the dependencies of images used by *Docker Actions*. As future work, we intend to perform an in-depth study of such Docker images, *Dockerfiles* and their corresponding commands, with the aim to detect the dependencies of *Docker Actions*. Investigating the package dependencies within Docker images enables practitioners to enhance the provisioning and deployment of *Docker Actions*. It can address potential concerns related to maintainability while also promoting the stability of the workflow for users of *Docker Actions*.

7. Threats to validity

Our study adopts the structure suggested by Wohlin et al. [39] to address the primary threats to validity. Threats to *external validity* relate to whether the results can be applied or extended beyond the specific scope of this study. One such threat was our choice to focus the analysis solely on Actions distributed through the GitHub Marketplace. Both the study conducted by Saroar and Nayebi [15] and our

own observations highlighted the existence of numerous Actions developed on GitHub repositories that were not published on the GitHub Marketplace. Another threat to external validity relates to filtering the Actions with at least 10 stars, in order to exclude personal or experimental Actions [35]. Moreover, we exclude around 100 Actions with more than 10 stars that appear in the GitHub Marketplace but produce an error when attempting to load detailed information for them, such as *c-documentation-generator*[5]. This implies that our findings may not be universally applicable to all Actions, as filtered and unpublished Marketplace Actions might possess characteristics that differ from those that we analysed.

Threats to *construct validity* discuss the connection between the theoretical basis of the experiment and the observed results. Our research utilises quantitative observations and rigorous methods in software repository mining, minimizing the impact of threats to construct validity.

Threats to *internal validity* address the choices and internal factors within the study that have the potential to influence the observations we have made. A potential threat to internal validity arises from the method we used to identify releases of Actions. As described in Section 4, we utilised the "releases" feature of the GitHub API to determine when Actions had been updated. This introduces a potential limitation, since not all Actions depend on the release management system of Github; some may distribute through other mechanisms such as *tags, git commits, or branches*. However, there is currently no accurate method to identify which git tags and branches correspond to releases, except by using GitHub's release management system.

Threats to *conclusion validity* concern the extent to which the conclusions drawn from our analysis are reasonable. Since our conclusions mostly state quantitative observations, it is unlikely that they will be influenced by such threats.

8. Conclusion

Since its introduction in 2019, GitHub Actions has become the *de facto* CI/CD automation service for GitHub repositories. This preliminary quantitative study aimed to understand the ecosystem of GitHub Actions and its dependency on external, third-party components such as Docker images and npm packages. Through an analysis of the GitHub development repositories of 2,817 Actions distributed through the GitHub Marketplace, we aimed to explore the characteristics of their dependencies. To do so, we distinguished between three types of Actions: those that are developed using JavaScript, those that are developed using Docker, and the so-called composite Actions.

We observed that, while most Actions are developed using JavaScript, a significant proportion of Actions are based on Docker. Moreover, after GitHub decided to introduce the mechanism of *composite Actions*, such Actions seem to be slowly replacing *Docker Actions*. Considering the fact that composite Actions can depend on other Actions, this might lead to more (transitive) dependencies in the GitHub Actions ecosystem.

JavaScript Actions have many dependencies towards npm packages, leading to a huge number of transitive dependencies that are deeply nested in the dependency tree. This makes JavaScript Actions considerably exposed to potential maintainability issues associated with their dependencies. It could become increasingly challenging to identify and resolve such issues, similar to what has been observed for JavaScript projects and packages [21, 25, 26, 40, 41].

We observed that Docker Hub is the most popular container registry for *Docker Actions* and the predominant used images by *Docker Actions* are associated with programming languages and operating systems. The GitHub container registry is another container registry that is commonly used by Action YAML files.

As future work, we intend to assess the impact of maintainability issues such as dependency management, security vulnerabilities, compatibility issues, outdated or obsolete components on the GitHub Action ecosystem due to its cross-ecosystem dependencies to Docker images and npm packages.

Acknowledgments

This research is supported by the Fonds de la Recherche Scientifique - FNRS under grant numbers PDR T.0149.22 and F.4515.23.

References

[1] Snyk, State of open source security 2022, https://snyk.io/reports/open-source-security/, 2022. [Online; accessed on May 1, 2023].

[2] 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

[5] https://github.com/marketplace?type=actions&query=c-documentation-generator+

in Computing Systems, 2011, pp. 3151–3160. doi:10.1145/1978942.1979409.

[3] L. Dabbish, C. Stuart, J. Tsay, J. Herbsleb, Social coding in GitHub: Transparency and collaboration in an open software repository, in: International Conference on Computer Supported Cooperative Work (CSCW), ACM, 2012, pp. 1277–1286. doi:10.1145/2145204.2145396.

[4] GitHub, Octoverse 2022: The state of open source software, https://octoverse.github.com/2022/developer-community, 2022. [Online; accessed 1 April 2023].

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

[6] 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, 2021. doi:10.1109/SANER53432.2022.00084.

[7] M. Wessel, T. Mens, A. Decan, P. Rostami Mazrae, The GitHub development workflow automation ecosystems, in: Software Ecosystems: Tooling and Analytics, Springer, 2023. doi:10.48550/arXiv.2305.04772.

[8] 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.

[9] 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 (2023) 52. doi:10.1007/s10664-022-10285-5.

[10] 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. doi:10.1109/MSR52588.2021.00054.

[11] 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. doi:10.1109/QRS-C55045.2021.00163.

[12] P. Valenzuela-Toledo, A. Bergel, Evolution of GitHub Action workflows, in: International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2022. doi:10.1109/saner53432.2022.00026.

[13] 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.

[14] 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.

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

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

[17] W. B. Frakes, K. C. Kang, Software reuse research: status and future, IEEE Transactions on Software Engineering 31 (2005) 529–536. doi:10.1109/TSE.2005.85.

[18] 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.

[19] 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. doi:10.1145/3106237.3106267.

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

[21] 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.

[22] 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 (2022) 1–45. doi:10.1007/s10664-022-10154-1.

[23] 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.

[24] H. H. Thompson, Why security testing is hard, IEEE Secur. Priv. 1 (2003) 83–86.

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

[26] 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, International Conference on Software Engineering (ICSE) (2022) 672–684. doi:10.1145/3510003.3510142.

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

[28] 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 (2021). doi:10.1109/TSE.2021.3112204.

[29] 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 (2021) 1–56. doi:10.1145/3447245.

[30] 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.

[31] 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 (2019) e2157. doi:10.1002/smr.2157.

[32] 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.

[33] 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.

[34] 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. doi:10.1109/ICSE.2015.140.

[35] 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.

[36] R. Shu, X. Gu, W. Enck, A study of security vulnerabilities on docker hub, Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (2017). doi:10.1145/3029806.3029832.

[37] J. Gummaraju, T. Desikan, Y. Turner, Over 30% of official images in docker hub contain high priority security vulnerabilities, Technical Report (2015).

[38] A. Zerouali, T. Mens, G. Robles, J. M. Gonzalez-Barahona, On the relation between outdated docker containers, severity vulnerabilities, and bugs, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019, pp. 491–501. doi:10.1109/SANER.2019.8668013.

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

[40] I. S. Makari, A. Zerouali, C. D. Roover, Prevalence and evolution of license violations in npm and rubygems dependency networks, in: International Conference on Software Reuse, 2022. doi:10.1007/978-3-031-08129-3_6.

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