

University of Mons
Faculty of Science
Computer Science Department

Fine-grained Exploration of the GitHub Workflow Ecosystem

Director : Dr. Tom MENS

Master's thesis written by
Laurent BOSSART

in order to obtain the grade of
Master in Computer Science



Academic year 2021 - 2022

ACKNOWLEDGEMENT

I want to address my deep regard and gratitude to Tom Mens for his guidance and valuable feedback throughout this work. I hope that future students will find in him the passion for computer sciences that I found in him.

I would like to express my heartfelt thanks to my family for all their encouragement throughout my studies. Without them, this work would never have been possible. I thank, with all my heart, Guillaume Szczepanski for all his support and friendship. I would also like to send a special thanks to Florent Huylenbroek, whose encouragement and advice allowed me to improve. Finally, I can't do without thanking Delphine Bossart for her precious help.

I would also like to extend a special thanks to the professors of the Faculty of Science at the University of Mons. They were always encouraging and pushed me to go further.

ABSTRACT

The topic of this master's thesis takes place within the general context of distributed open source software development through GitHub¹. GitHub being an online collaborative platform based on the distributed Git² version control system. The purpose of GitHub is to allow developers to share code and to allow collaboration between developers. This platform provides a mechanism to automate the creation and configuration of development workflows through the use of GitHub Actions³. An Action is a piece of code allowing repository maintainers to write custom code that interacts with their repository in many ways, e.g. an action can check code quality, look for security vulnerabilities, interact with continuous integration systems, and many more. Actions representing a workflow need to be specified in a YAML file stored in the *.github/workflows/* directory in repositories of projects on GitHub. Actions can be triggered in different ways, including when a push or pull requests occurs on the repository. Once triggered, the Action will execute the content of its YAML file, configured by the developer.

A study conducted by Golzadeh et al. [1] indicated that GitHub Actions has become one of the most popular tools to automate the continuous integration workflow. Continuous integration (CI) is a software integration strategy aiming to integrate the work of each member of a team of developers frequently [2]. Therefore, it is useful to conduct a quantitative empirical study about GitHub Actions in order to have a better understanding of their use and evolution. This work presents such a study aiming to gain a deeper understanding of how the GitHub Actions ecosystem is structured and how it evolves over time. Historical data about GitHub Actions available on the GitHub Marketplace⁴ has been collected

¹<https://github.com/>

²<http://git-scm.com/>

³<https://github.com/features/actions>

⁴<https://github.com/marketplace>

and observed in order to make observations about their use and evolution.

First, an observation has been made about the growth of the GitHub Marketplace overtime, the evolution of Actions, and the proportion of Actions in 18 distinct categories. Using the content of YAML files in the *.github/workflows/* directory of selected GitHub repositories, the number of Actions that were the most used in workflow, and the number of distinct Actions that were used by these GitHub repositories was identified. Then, the most popular Actions were identified along with the proportion of open and closed issues for each observed Action repository. Furthermore, the GitHub Marketplace indicate if the owner of an Action is verified, meaning that GitHub verified the ownership of their domain, confirmed their email address, and are using two-factor authentication for their organization. Therefore, an observation has been made to identify the proportion of Actions owned by verified users. Finally, since developers have the possibility to trigger Actions in different ways, an observation has been made to identify the triggers that are the most commonly used.

The observations indicated that the number of Actions available on the GitHub Marketplace is growing, the majority of observed Actions are being updated every month, some Marketplace categories are more popular than others, and that developers are interested to use Actions when push and pull requests are made on their repositories. These observations give a more global vision of the use of GitHub Actions by developers, and how their ecosystem evolves over time.

Keywords: GitHub, GitHub Actions, Development Workflow, Continuous Integration, Software Repositories, Automation

Contents

List of Figures	vi
List of Tables	viii
1 Introduction	1
2 State of the Problem	6
2.1 GitHub Actions	6
2.2 GitHub Actions and Other CI/CD Solutions	18
2.2.1 CircleCI	18
2.2.2 Travis CI	19
2.2.3 Observations	19
2.3 State of the Art	20
2.3.1 Kinsman et al.'s Article	20
2.3.2 Golzadeh et al. Article	23
2.3.3 Conclusion	26
3 Realization	27
3.1 Programming Language	27
3.2 Data Extraction	34
3.2.1 Python Selenium Library	36

3.2.2	Python <i>requests</i> Library	37
3.2.3	Extraction	37
3.3	Data Storage	43
3.4	Data Analysis	47
4	Results	51
4.1	RQ1: How Fast Does the GitHub Marketplace Grows Over Time? .	51
4.2	RQ2: Which Categories of Actions Are Most Commonly Proposed on the Marketplace?	56
4.3	RQ3: How Fast Are Actions Evolving?	60
4.4	RQ4: Do Workflows Use Multiple Actions?	69
4.5	RQ5: What Are the Most Popular Actions?	77
4.6	RQ6: Are There Many Actions With Known Unresolved Issues? . .	84
4.7	RQ7: Which Proportion of Actions Are Developed by Verified Users?	89
4.8	RQ8: How Are Actions Triggered?	93
5	Conclusion	96
	Bibliography	98

List of Figures

2.1	Workflow Run.	9
2.2	GitHub Marketplace.	11
2.3	Information about the “first interaction” Action.	12
3.1	Overview of the information that can be found on a GitHub Marketplace page for an Action.	29
3.2	Overview of the information that can be found on a GitHub repository page for an Action.	30
3.3	Overview of the information about the contributors of an Action that can be found issuing a request to the <code>https://api.github.com/repos/<username>/<repository>/contributors</code> endpoint of the GitHub REST API.	31
3.4	Example of information about issues of a GitHub repository extracted using the GitHub GraphQL API.	32
3.5	Second example of information about issues of a GitHub repository extracted using the GitHub GraphQL API.	33
3.6	Overall number of Actions on the Marketplace.	42
3.7	Graphical representation of the database used to save the data.	46
3.8	Number of Actions per categories - <i>matplotlib</i>	48
3.9	Number of Actions per categories - <i>seaborn</i>	49

4.1	Weekly number of Actions on the Marketplace.	53
4.2	Number of new Actions added to and removed from the Marketplace.	54
4.3	Weekly number of Actions in the “ides” category.	55
4.4	Weekly number of Actions in the “learning” category.	55
4.5	Number of Actions by category on the Marketplace on July 31, 2022.	57
4.6	Box plot for the distribution of Actions per category.	59
4.7	Distribution of the time between major, minor, and patch releases, respectively.	67
4.8	Distribution of the time between releases.	68
4.9	Distributions of the number of Actions per workflow file.	71
4.10	Distribution of the occurrences of Actions in the overall workflow files.	76
4.11	Actions listed by popularity on the Marketplace.	80
4.12	Example of issue with comments.	86
4.13	Distribution of seconds between the moment an issue is open and closed.	87
4.14	Distribution of seconds between the moment an issue is open and the current date.	88
4.15	Verified badge on the Marketplace.	90
4.16	Verified badge on one Action page.	91

List of Tables

2.1	Differences between GitHub Actions, Jenkins, and Travis CI.	20
3.1	GitHub API URLs.	40
3.2	Useful methods for the statistic analysis.	50
4.1	Weekly number of Actions on the Marketplace.	53
4.2	Number of times an Action is added to and removed from a category.	55
4.3	Values for the number of Action for each category on July 3, 2022.	59
4.4	Non-exhaustive list of examples of accepted release tag formats.	63
4.5	t-test: p-values.	65
4.6	Number of updates by type of release.	66
4.7	Mann-Whitney U test: p-values.	66
4.8	Seconds between major releases.	67
4.9	Seconds between releases.	68
4.10	Number of Actions per workflow file.	71
4.11	Ten most used Actions in workflow files.	73
4.12	Comparison between Kinsman et al. [3] top ten and the current top ten.	75
4.13	Actions occurrences in the overall workflow files.	76
4.14	Weights for each events	79
4.15	The ten most popular Actions.	83

4.16	Most popular categories.	83
4.17	Most popular owners.	83
4.18	Distribution of seconds between the moment an issue is open and closed.	87
4.19	Distribution of seconds between the moment an issue is open and the current date.	88
4.20	Number of unverified, overall number, and percentage of unverified Actions/users on the Marketplace.	91
4.21	Number of verified Actions per category.	92
4.22	Most used triggers in workflow files.	94

Chapter 1

Introduction

GitHub is a website where developers can store and update their software code. It is based on git, and is one of the most important sources of software on the Internet [4, 5]. The GitHub state of the Octoverse¹ for the year 2021 indicated that there was 61M+ new repositories on GitHub, and a total of 73M+ developers using the platform. GitHub is much more than a simple website, it is a social coding site, meaning that it implements a social network where developers can share their work with other interested developers [6]. It allows users and developers to open issues, make pull requests, and make forks among other things. By making a fork, developers copy the main repository allowing them to make their own changes on this copy. Then, they can make a pull request on the original repository in order for the developer of this repository to review, and maybe integrate, the new functionalities/changes made on the copy. This approach gives the opportunity to each developer to improve an already existent project, but it increases the workload for repository maintainers [4].

In order to reduce the workload of developers, GitHub introduced GitHub Actions in November 2019 to allow developers to automate repetitive tasks in their

¹<https://octoverse.github.com/>

workflows, like checking if the code builds [4]. GitHub Actions are designed to automate parts of the workflow, with the aim to increase both productivity and quality [7]. GitHub Actions can be triggered in different ways, like pull requests, issue opening, etc. Actions can be easily found on the GitHub Marketplace so that developers can use them in their repositories, and GitHub supports continuous integration (CI) and continuous deployment (CD) through the use of GitHub Actions. Referring to the accelerate state of DevOps 2021 [8], organizations mastering CI/CD deploy 973 times more often and have a lead time² 6,570 times faster than organizations not using CI/CD. Furthermore, the GitHub resource about CI/CD[9] indicates that there are multiple benefits of CI/CD among which a better development velocity (developers are committing smaller changes more often), a better stability and reliability (an automated and continuous testing ensures that codebases remain stable and release-ready at any time). GitHub Actions are not the only solution to set up CI on a project. Some well-known examples are Travis CI³, Jenkins⁴, and CircleCI⁵. The main difference between these tools and GitHub Actions is that GitHub Actions are directly integrated to GitHub. As an example, in order to use Jenkins with a GitHub project, it is needed to set up and maintain a server with Jenkins installed on it. Another example is Travis CI which offers a cloud-based solution, but needs to be setup with GitHub. With GitHub Actions, such set up is not required.

As far as we know, the study by Kinsman et al. [3] and the one by Golzadeh et al.[1] are the only two studies that deal with GitHub Actions. In this master's thesis, a set of GitHub Actions will be analyzed in order to have a broader view on how they evolve, and how they are used. To achieve this, the following research questions aim to be answered:

²Time to go from code committed to code running in production.

³<https://www.travis-ci.com/>

⁴<https://www.jenkins.io/>

⁵<https://circleci.com/>

RQ1: *How Fast Does the GitHub Marketplace Grows Over Time?* This RQ aims to understand how fast the GitHub Marketplace for the Actions is growing over time. To do this, data about Actions that were available on the Marketplace will be collected at different time stamps. The overall number of Actions and the number of Actions for each category will be analyzed on the different sets of data.

As an answer to this question, it has been observed that the number of Actions added on the Marketplace was higher than the number of Actions deleted, implying an overall increase in the number of Actions. This increase has also been observed by Golzadeh et al. [1]. Two categories out of 18 experienced a loss in their number of Actions between the first and last collect of data. An explanation to this loss might be the obsolescence of Actions and their removal from the Marketplace by their owner, or because the owners “moved” them to another category.

RQ2: *Which Types of Actions Are Most Commonly Proposed on the Marketplace?* In this RQ the number of Actions for each category will be analyzed in order to understand which types of Actions were most likely to be sought by developers. Knowing what categories have the highest number of Actions might indicate which categories of Actions are the most needed by developers. We observed 67.5% of all observed Actions were belonging in seven distinct categories. It might indicate that developers were more interested in categories such as *publishing* and *code quality*. These categories had already been observed by Kinsman et al. [3].

RQ3: *How Fast Are Actions Evolving?* This RQ aims to determine how fast the Actions are getting new releases. A fast evolution might indicate that the studied Actions are well maintained, in contrast to a slow evolution which might indicate that they are not well maintained. The time elapsed between each release of Actions will be analyzed, and the type of release will be determined. A release can be a patch, a minor release, or a major release.

It has been observed that patches were released more often than minor and major releases, with major releases being released less often than minor releases. This observation implied that Actions were evolving fast.

RQ4: *Do Workflows Use Multiple Actions?* In order to understand the number of distinct Actions used by GitHub repositories, the content of GitHub repositories using GitHub Actions will be analyzed. This information gives us a first idea about the Actions that are the most used.

The majority of workflow files were using less than three Actions, and very few workflow files were not using Actions available on the Marketplace at all. Also, similarities between the top ten of the most used Actions observed by Kinsman et al. [3] and the top ten we observed were found.

RQ5: *What Are the Most Popular Actions?* Based on the data of the 8,114 Actions that were gathered, a formula has been defined to represent the popularity of an Action. Using this formula, the most popular Actions were identified. Having this information will give us a first idea of what the Actions are most useful for. Also, the categories containing the highest number of popular Actions were considered has been the most popular ones. Finally, the most popular owners were identified.

RQ6: *Are There Many Actions With Known Unresolved Issues?* The time between the creation and the resolution of issues has been computed to determine if an Action is well maintained or not. Knowing this proportion will give us an idea on the number of Actions that can potentially cause problems when they are used.

A minority of Actions were considered as well maintained. Also, the time between the creation and resolution of issues suggests that either an issue is quickly closed, or it takes a long time to close it.

RQ7: *Which Proportion of Actions Are Developed by Verified Users?* The

proportion of Actions developed by verified users will give an indication on the percentage of trustful Actions present on the Marketplace.

The majority of Actions are owned by unverified users. This implies that users are not seeing verification as a strong trust factor but might consider the popularity of an Action as a stronger metric.

RQ8: *How Are Actions Triggered?* Knowing how Actions are triggered will give us a better understanding of how developers are using Actions in their workflows.

It was observed that Actions were essentially used when push and pull requests occurs on a repository.

This thesis is structured as follows. Chapter 2 gives a state of the art for the subject of this study. It discusses GitHub Actions and gives a summary of the two previous known articles dealing with GitHub Actions. Chapter 3 explains the technical aspects of this study, i.e. how the data as been collected, stored, and analyzed. Chapter 4 aims to answer the question asked in the introduction. Finally, Chapter 5 concludes.

Chapter 2

State of the Problem

This chapter gives a description of what are GitHub Actions, why they are useful, and how we can use them. It also gives an explanation on how to find and create them. GitHub Actions being a continuous integration and continuous delivery (CI/CD) solution, there is also an explanation about other CI/CD solutions, and the differences between them and the GitHub Actions.

Section 2.1 presents GitHub Actions. Section 2.2 explores the differences between GitHub Actions and other CI/CD solutions. Finally, section 2.3 discusses scientific articles that have empirically studied GitHub Actions.

2.1 GitHub Actions

This section gives more detailed information about what are GitHub Actions. GitHub Actions are a CI/CD service released to the public in November 2019. Even though there were already CI/CD services on the market, GitHub Actions quickly became popular [1]. Golzadeh et al. [1] made a few hypotheses about this popularity, including the fact that GitHub Actions are fully integrated with GitHub, or because GitHub Actions comes with a large marketplace allowing repository main-

ainers to select the Actions they need. Based on the official documentation about GitHub Actions provided by GitHub [10], here is a description of this service.

GitHub Actions are used to automate the development workflow, and allow the automation of tasks that can be triggered in various ways. Triggers are events that can occur on a repository, e.g. a newly created pull request, a new issue, commits, creation or modification of discussions, forks, creation/modification of repository wiki pages, and more¹. GitHub Actions are made to make the building, testing, and deployment of projects faster. The use of GitHub Actions is quite simple. The owner of a GitHub repository has to create a YAML file (representing a workflow) in the `.github/workflows/` directory, following the correct syntax. Those files can be configured to run a series of commands after one or more specific events occurred. A workflow file can contains none or multiple Actions, and developers can create their own Actions. The following sample of code represents an example of a workflow file taken from the *pykeen* repository², and the file is called *pr_welcolme.yml*.

```
1 name: Welcome
2
3 on:
4   pull_request:
5     types:
6       - opened
7     branches:
8       - master
9
10  jobs:
11    welcome:
12      runs-on: ubuntu-latest
13      steps:
14        - uses: actions/first-interaction@v1
15          with:
```

¹<https://docs.github.com/en/actions/reference/events-that-trigger-workflows>

²<https://github.com/pykeen/pykeen>

```

16     repo-token: \${{ secrets.GITHUB_TOKEN }}
17     pr-message: |-
18         Congrats on making your first Pull Request and thanks for taking
19         the time to improve PyKEEN!

```

Here is a description of the different keywords used in this sample:

- **name:** Defines the name of the workflow. This is the name that will be displayed on the repository’s “actions” page (<https://github.com/pykeen/pykeen/actions> in our example). Here, the name is “Welcome”.
- **on:** Defines the event(s) that will trigger this workflow. Here, this workflow will run in the context of pull requests.
- **types:** Gives more control over when the workflow should run. Here, the workflow will run when pull requests are opened.
- **branches:** Defines the branch(es) on which the event will be triggered. Here, it mean the workflow will run if the pull requests types occurs on the branch called “master”.
- **jobs:** Defines the steps of the workflow. Each jobs are running in parallel by default. Here, there is only one job called “welcome”.
- **runs-on:** Defines the runner on which the job will run. Here, the job will run on a Ubuntu (latest version) virtual machine.
- **steps:** Represents the sequence of tasks that the job will run. Here, there is one task defined by the “uses” keyword.
- **uses:** Allows to select an Action as part of the step in a job. In this case, the Action used is “actions/first-interaction@v1”. “actions” is the owner of the Action, “first-interaction” is the name of the Action, and “@v1” means it is wanted to use the version with major version one of the Action.
- **with:** Is the input for the parameters of the Action. Here, the first parameter is *repo-token*, and has for value a secret variable called GITHUB-TOKEN.

Secret variables can be defined on every GitHub repository by going on the settings of the repository `Secrets` `Actions`. The second parameter is *pr-message* and contains a string that will be passed as parameter to the “first-interaction” Action.

A workflow run can be observed on the following URL of a GitHub repository: “`https://github.com/[owner_of_the_repository]/[name_of_the_repository]/actions/workflows/[name_of_the_workflow_file].yaml`”. As an example, figure 2.1 illustrates such a run.

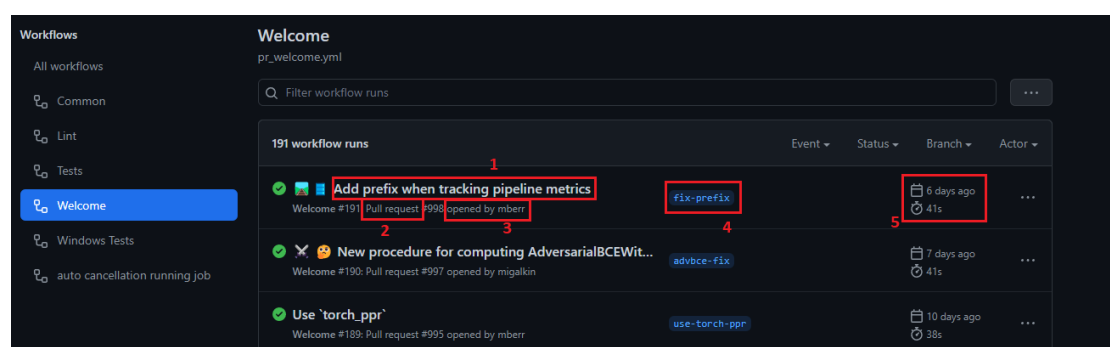


Figure 2.1: Workflow Run.

On this figure, frame 1 is the name of the event, frame 2 is the event that triggered the workflow (here a pull request), frame 3 is the type of event and the user that initiated it (here the user *mberr* opened a pull request), frame 4 is the branch from which the event has been initiated (here it is wanted to make a pull request on the master branch from the *fix-prefix* branch). Finally, frame 5 gives an indication on when the workflow has been triggered and the time it took to complete.

A workflow has the particularity of being reusable under two conditions, as described by the official documentation provided by GitHub [11]. A caller workflow can use a called workflow if they both are on the same repository (in the *.git/workflows* directory). Otherwise, the called workflow must be stored in a

public repository, and the developer's organization must allow them to use public reusable workflows. There are also some limitations. A reusable workflow can not call other reusable workflows. A reusable workflow stored within a private repository can only be used by workflows within the same repository, any environment variables set in an *env* context defined at the workflow level in the caller workflow are not propagated to the called workflow, and the *strategy* property is not supported in any job that calls a reusable workflow. To create a reusable workflow, the official documentation indicates that the value for *on* must include *workflow_call*, as illustrated by the following sample

```
1 name: Welcome
2
3 on:
4   workflow_call:
```

To call a workflow from another workflow, a developer has to use the *uses* keyword in the same way as explained before. The syntax for reusable workflows in the same repository is “./.github/workflows/[file name]”, and “[owner]/[repo]/.github/workflows/[file name]@[ref]” for reusable workflows in public repositories. For reusable workflows in public repositories, the [ref] can be a commit SHA, a release tag, or a branch name. The documentation indicates that using the commit SHA is the safest for stability and security. The following sample gives an overview of how a developer can perform this.

```
1 jobs:
2   call-workflow-1-in-local-repo:
3     uses: octo-org/this-repo/.github/workflows/workflow-1.yml@172239021f7ba04fe
4     7327647b213799853a9eb89
5   call-workflow-2-in-local-repo:
6     uses: ./.github/workflows/workflow-2.yml
7   call-workflow-in-another-repo:
```

```
uses: octo-org/another-repo/.github/workflows/workflow.yml@v1
```

It is to be noted that a large variety of Actions are available on the GitHub Marketplace, as shown in figure 2.2. On this figure, we can see that 13K Actions were available within different available categories (*API management*, *Chat*, *Code quality*, and so on). A search bar can be used to find specific Actions, and some Actions (along with their name, category, a description and the number of stars) can be observed. This marketplace makes the sharing of Actions easy, and using GitHub Actions makes code reviews, code quality analysis, etc. more easier.

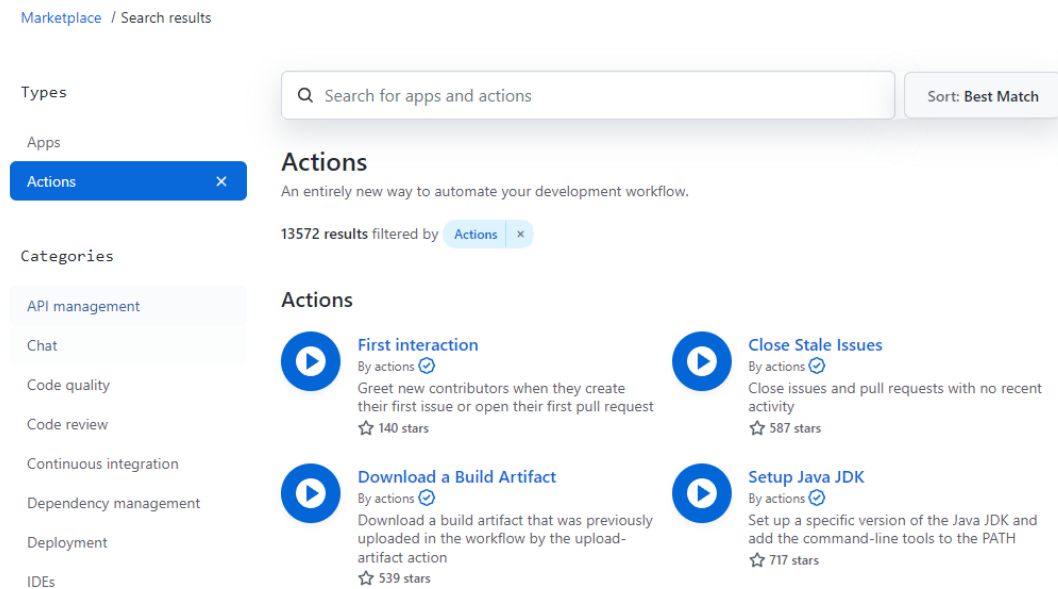


Figure 2.2: GitHub Marketplace.

When clicking on an Action name, we can access a variety of information about the Action. This is illustrated by figure 2.3, with the Action called “first interaction” as example.

The screenshot displays the GitHub Actions marketplace page for the 'First interaction' action. The page is annotated with red boxes and numbers 1 through 9, highlighting specific features:

- 1:** The action's name 'First interaction' and its current version 'v1.1.0' (labeled 'Latest version').
- 2:** The description: 'An action for filtering pull requests and issues from first-time contributors.'
- 3:** The 'Usage' section, which includes a link to 'See action.yml' and a code block showing the configuration for using the action in a workflow.
- 4:** The 'Use latest version' button.
- 5:** The 'Verified creator' badge, indicating that GitHub has verified the action was created by actions.
- 6:** The 'Stars' section, showing the action has 391 stars.
- 7:** The 'Contributors' section, displaying a grid of user avatars.
- 8:** The 'Categories' section, showing the action is categorized under 'Utilities'.
- 9:** The 'Links' section, providing links to the action's repository ('actions/first-interaction'), open issues (30), pull requests (18), and a report abuse link.

Figure 2.3: Information about the “first interaction” Action.

On the previous figure, a developer can find a set of information about the Action:

1. The name of the Action, along with its current version.
2. A description of what is the Action doing.
3. An example on how to use the Action.
4. The list of available versions.
5. An indication of whether the user is verified by GitHub or not.
6. The number of stars the GitHub repository of the Action was granted.

7. An overview of the contributors.
8. The GitHub marketplace categories in which this Action falls.
9. Some links to the GitHub repository of the Action, along with a link to report any kind of abuse.

Here are some definitions obtained from the official documentation of GitHub [10] to fully understand what are GitHub Actions:

- **Workflows:** A workflow can be seen as an automated process that can be configured and will run one or more jobs. A workflow is defined by a YAML³ file, and will run when triggered by an event occurring in the repository, such as a release, a pull request, or commits, and more (see Events below). They can also be triggered manually, or at a defined schedule.

Workflows are defined in a specific directory in a GitHub repository (*.github/workflows/*). This directory can have multiple workflows that can perform different tasks. The following example has been presented on the following link <https://github.community/t/depend-on-another-workflow/16311>. Assuming that the developer has enabled protection settings on their master branch to only allow merging pull requests whenever their CI workflow have run successful, they can create a secondary workflow that only triggers when the pull request is merged. Thus, the workflow files syntax allow one workflow to depend on another one, this situation is represented by the following workflows.

```
1  name: Workflow_1
2
3  on:
```

³<https://yaml.org/>

```

4      pull_request:
5        branches:
6          - master
7
8      jobs:
9        testing:
10         runs-on: ubuntu-latest
11
12        steps:
13          ...

```

```

1      name: Workflow_2
2
3      on:
4        pull_request:
5          types: [closed]
6          branches:
7            - master
8
9      jobs:
10       deploy:
11         runs-on: ubuntu-latest
12         if: github.event.pull_request.merged
13         steps:
14           - name: checkout
15             if: github.event.pull_request.merged
16             uses: actions/checkout@v1
17             with:
18               ref: master
19         ...

```

The first workflow (Workflow_1) runs on new pull requests. The second one runs when a pull request is merged through the use of the “if” expression. This way, Workflow_2 will only execute when Workflow_1 has completed and the pull request has merged.

For more information about workflows, the reader may consult <https://docs.github.com/en/actions/using-workflows>.

- **Events:** An event that triggers a workflow run is called an event. For example, an activity could be a pull request, a commit, a new release, etc. To specify that a workflow is triggered by an event, developers may use the “on” keyword, as shown in the following sample.

```
1  name: Example
2
3  on:
4    gollum
5    ...
```

This workflow will be triggered when someone creates or updates a Wiki page in the repository.

The complete list of events is available on the following link <https://docs.github.com/en/actions/reference/events-that-trigger-workflows>.

- **Runners:** A runner is a server that runs the workflows when they are triggered. A runner can only run one single job at a time. GitHub provides Linux, Windows, and macOS runners. Each workflow run always executes in a fresh virtual machine. It is possible to use self-hosted runners instead of the ones proposed by GitHub. To use a runner, the developer has to use the “runs-on” keyword. The following example illustrates how to use it.

```
1  name: Example
2
3  on:
4    push
5
```

```
6   job:
7     example_job
8     runs-on: ubuntu-latest
9     ...
```

Here, the job called “example_job” will run on a virtual machine using the latest version of Ubuntu.

For more information about runners, see <https://docs.github.com/en/actions/hosting-your-own-runners> and <https://docs.github.com/en/actions/using-github-hosted-runners>.

- **Jobs:** A step in a workflow is called a job. All jobs in a workflow are executed on the same runner. A job can be a shell script that will be executed, or an Action that will be run. Jobs are executed in order and are dependent on each other. Since jobs are executed on the same runner, they can share data from one job to another. By default, jobs run in parallel with each other.

It is also possible to use a matrix strategy in a single job. This matrix strategy allows the owner of the workflow to use variables in a job and create multiple job runs based on these variables. The following code sample shows an example of this feature.

```
1   name: Matrix
2   ...
3   jobs:
4     example_matrix:
5       strategy:
6         matrix:
7           os: [ubuntu-18.04, ubuntu-20.04]
8           version: [10, 12, 14]
9       runs-on: ${ matrix.os }
10      steps:
11        - uses: actions/setup-node@v3
```

```
12         with:
13             node-version: \${{ matrix.version }}
14     ...
```

This example will run a jobs using the variables defined in the matrix. The first job will run on “ubuntu-18.04” with the node version 10, the second one will run on “ubuntu-18.04” with the node version 12, etc. In fine, their will be a job created for each combination of variables. This example could be used to ensure platform compatibility for different versions.

For more information about jobs, refer to <https://docs.github.com/en/actions/using-jobs>, and for more information about matrix, consult <https://docs.github.com/en/actions/using-jobs/using-a-matrix-for-your-jobs#example-using-a-single-dimension-matrix>.

- **Actions:** An Action is a custom reusable application that performs a complex but frequently repeated task. Actions help reduce the repetitive code in workflow files. It is possible for a developer to write its own actions⁴, or to find Actions to reuse from the GitHub Marketplace.

To create a custom Action, a developer has to choose between running this Action directly on a machine or in a Docker container. It is to be noted that Actions running on a Docker container are using Linux, while Actions running directly on a machine can run on Windows, macOS and Linux. There are three types of Actions: Docker container, JavaScript, and Composite Actions. A composite Action allows you to combine multiple workflow steps within one Action.

For more information about the actions, see <https://docs.github.com/en/actions/creating-actions>.

⁴<https://docs.github.com/en/actions/creating-actions>

2.2 GitHub Actions and Other CI/CD Solutions

This section compares GitHub Actions with other CI/CD tools, namely CircleCI⁵ and Travis CI⁶, which were described as being part of the seven most popular CI services by Golzadeh et al. [1]. The comparison between those two CI services and GitHub Actions is made in line with Golzadeh et al. [1] observations. Indeed, they pointed out that Travis and CircleCI were (by far) the most used CI services before the arrival of GitHub Actions, and these two CI services are among of the oldest on the market (they were both launched back in 2011), and are older than GitHub Actions that were launched in 2019. This section aim to understand which characteristics of GitHub Actions made them become more popular at the expense of the other CI/CD tools. A comparison between GitHub Actions, CircleCI and Travis CI has been made in table 2.1.

2.2.1 CircleCI

Information about CircleCI were found on the CircleCI official documentation [12]. CircleCI is a continuous integration and continuous delivery platform. It can be used to implement DevOps practices, and was launched back in 2011. When using CircleCI, workflows are running on CircleCI's cloud, but it is possible to use a self-hosted server. CircleCI can be used with BitBucket and GitHub. CircleCI uses a system called *orbs*⁷ to allow reusable packages. There is, at the time of this writing, 3,176 orbs available on the following URL <https://circleci.com/developer/orbs>. CircleCI has a free plan with some limitations such as a limit of 6K build minutes per month. Even if a set up is required in order for CircleCI to work with GitHub and BitBucket, there is no installation needed while using their

⁵<https://circleci.com/>

⁶<https://www.travis-ci.com/>

⁷<https://circleci.com/orbs/>

cloud solution. Workflow runs has to be checked via the CircleCI user interface.

2.2.2 Travis CI

Travis CI is a CI/CD platform that as been launched back in 2011. With Travis CI developers are able the quickly build, test, and deploy code [13]. The workflows are running on Travis CI's cloud, and Travis CI requires that the owner of the repository to make some set up. Travis CI works with GitHub, BitBucket, and other platforms. Travis CI is free and open source for open source projects, otherwise it becomes a paid service. Workflow runs has to be checked via the Travis CI user interface.

2.2.3 Observations

GitHub Actions provides everything a developer using GitHub might need in order to integrate CI/CD in its workflow. There is no need of setting up a server, or checking results on a third party user interface. Everything has been made to integrate them easily, and everything is under the same roof making incompatibilities out of the equation. Wokrflow runs can either be executed on GitHub's cloud, or on self-hosted servers. There is a marketplace allowing developers to find the Action they need between a selection of 13,957.

These particularities could partially explain why GitHub Actions are becoming so much popular.

Criterion	GitHub Actions	Travis CI	CircleCI
Where executed	GitHub provides cloud runners, and developers can self-host a runner themselves.	Runs on Travis CI's servers. Must use Travis CI UI to check for runs, builds, etc.	Runs on CircleCI's servers, but possibility to have a self-hosted server. Must use CircleCI UI to check for runs, etc.
Compatibility	Compelled to use GitHub.	Can be used with any solution of choice, including GitHub, BitBucket, and more.	Can be used with GitHub and BitBucket.
Diversity	Marketplace that contains roughly 14K Actions.	Can import shared config files.	There is 3,176 orbs available.
Price	Freenium.	Freemium.	Freemium.
Installation	No installation needed.	Need some configuration to work with the repository.	Need some configuration to work with the repository.
Available since	2019	2011	2011

Table 2.1: Differences between GitHub Actions, Jenkins, and Travis CI.

2.3 State of the Art

This section gives a description of empirical studies that have already dealt with GitHub Actions. Two articles are presented, namely the article by Kinsman et al. [3] which is directly concerned with certain aspects of GitHub Actions, and the article by Golzadeh et al. [1] which is more concerned about the evolution of CI/CD tools in GitHub.

2.3.1 Kinsman et al.'s Article

The article written by Kinsman et al. [3] aimed to give a preliminary understanding on the effects of adopting GitHub Actions within ten months after its official launch. The content of this subsection is taken from this article.

The authors had three research questions:

1. How do OSS projects use GitHub Actions?
2. How is the use of GitHub Actions discussed by developers?

3. What is the impact of GitHub Actions?

To answer each of these questions, they started by gathering and analyzing some data. They proceeded in four steps.

The first one was the **project selection**. To select the projects, they used the GitHub project metadata of the RepoReaper⁸ data set, which contained 446,962 GitHub repositories classified as containing an engineered software project. They filtered these projects to keep the ones that adopted GitHub Actions at some point in their existence. To identify these projects, they used the GitHub REST API⁹ using a Ruby toolkit called Octokit.rb¹⁰. For every repository they gathered, they checked if it contains YAML files in the *.github/workflows/* directory. This filtering exposed 3,190 projects that could be used for the study.

Then, they **analyzed the use of GitHub Actions**. They analyzed previously filtered projects workflow files in order to determine their category, description, and whether the owner is verified by GitHub or not. To understand the evolution of GitHub Actions, they fetched the history of commits for every file in the workflow used by the selected projects. To do that, they looked at the history for changes linked to Actions, including the addition, deletion, and changes in configurations and versions.

For the **first research question**, they identified 3,190 active open-source projects having adopted at least one Action at the time. The data has been collected only 10 months after the launch of GitHub Actions. Among these projects, 708 different Actions were identified as being used. In order to classify them, they fetched data on the GitHub Marketplace and on the repositories of the Actions. By using the commit history of these repositories, they noticed that some repositories deleted, changed some parameters, or updated Actions. They pointed

⁸<https://github.com/RepoReapers/reaper>

⁹<https://docs.github.com/en/enterprise-server@3.2/rest>

¹⁰<http://octokit.github.io/octokit.rb/>

out that the most deleted Actions were also the most added ones.

Afterwards, they **categorized discussions about GitHub Actions**. In order to give an answer to the second research question, they fetched issues mentioning "github action" or "github actions" in the repositories. The issues had to have at least one comment and had to have been posted after the deployment of GitHub Actions. They found 209 issues that met the criteria, and analyzed them manually.

The **second question** showed how they categorized 209 issues based on the content of their discussions. They used six categories, including "GitHub Actions maintenance" and "Announcement GitHub Actions". Overall, discussions and frustrations about problems were compensated by the announcement of GitHub Actions implementation, requests of implementation, and the switching of CI/CD tools.

Finally, they made a **time series analysis** to answer the third research question. They collected longitudinal data for different variables so that they could compare how these variables evolved before and after the adoption of GitHub Actions. Examples of variables were: time elapsed since the first pull request, total number of commits, number of opened pull requests.

Then, in the **third research question**, they analyzed the repositories history and found that, after adopting GitHub Actions, repositories had a tendency to have more rejected pull requests, and less commits on merged pulls requests.

They identified that a small subset of the observed repositories were using Actions, and only 708 unique predefined Actions were being used within the workflows. Observing the issues related to GitHub Actions, they found that the majority of the discussions were positive, and GitHub Actions were met with an overall positive reception. Finally, they observed that, after the adoption of GitHub Actions, the number of commits of merged pull requests decreases and there were

also more monthly rejected pull requests.

The authors suggested a qualitative investigation of the effects of adopting GitHub Actions, and an expansion of their analysis for considering the effects of different types of Actions and activity indicators.

2.3.2 Golzadeh et al. Article

The article by Golzadeh et al. [1] consists of an empirical study that focuses on four research questions:

1. How did the CI/CD tools landscape evolve?
2. What are the most frequent combinations of CI/CD tools?
3. How frequently are CI/CD tools being replaced by an alternative?
4. How has the CI landscape changed since GitHub Actions was introduced?

The remainder of this section is based on the content of their article.

The first step of their study was to **extract the data**. They needed a large dataset containing thousands of repositories of diverse projects in order to analyze the use of CI/CD tools. They excluded from the dataset repositories created for experimental or personal reasons, or showing small traces of commits. The researchers have identified registries for reusable software packages, such as npm for JavaScript or Maven for Java, as good candidates.

They used the npm¹¹ (the largest software library and a software package manager for JavaScript¹²) API to make a list of all 1.6M+ scoped packages. On May 2021, they collected the data and observed that 803K npm packages were referencing a GitHub repository. They cloned 676K packages with an accessible GitHub

¹¹<https://www.npmjs.com/>

¹²<https://www.javascript.com/>

repository from this list. Also, they excluded 11,557 repositories that were forks of other repositories. In fine, their dataset consisted of 201,403 repositories.

Next, they established a list of 28 candidates CI/CD tools based on scientific articles, developers websites, and blog posts. Reading the documentation of these tools, they identified configuration files specific to each one. Based on this documentation, they were able to exclude tools that were configured via a user interface, and using configuration files with no fixed name. Finally, 20 CI/CD tools remained for further analysis, including Travis, GitHub Actions, and Jenkins.

Afterwards, they analyzed every commit of each of the 201,403 cloned repositories to determine if configuration files related to CI/CD tools were present. They identified 95,035 repositories with such files in it. They also identified problems with some GitHub repositories and excluded 60 repositories from the research.

The presence of configuration files was used to determine when CI/CD tools were added/deleted from a repository. They decided not to include in the study projects having only used CI/CD tools for a few days. Such projects represented 6,586 repositories.

Once the data was collected and cleaned, they answered the **first research question**. With this question, they sought to understand which CI/CD tools were used by repositories in their dataset, to what extent they are prevalent, and how this evolved over time. They noticed that the number of repositories using CI/CD tools has tended to increase over time. The more surprising phenomena they observed was the fast-paced evolution in the use of GitHub Actions. Within only 18 months of existence, GitHub Actions took the second place in terms of usage. This suggests that GitHub Actions dramatically changed the CI/CD tools landscape. Still in answer to this question, they observed a decrease in the number of repositories opting for some CI/CD tools, as well as a slower evolution of the adoption of other CI/CD tools. However, they noticed GitHub Actions was the

only tools which was always increasing in terms of usage.

For the **second research question**, they found that there was more CI/CD tools usage than number of repositories. This implied that some repositories used more than one CI, either at the same moment or at some point in their life span. The majority of repositories using multiple CI/CD tools included Travis, GitHub Actions and CircleCI. Those repositories represent 22,877 repositories, with the majority of them including the use of Travis, AppVeyor, GitHub Actions, and CircleCI. Anyhow, they noticed that the majority of CI/CD tools were not used simultaneously with another CI.

Then, in the **third research question**, they stated that when a repository uses multiple CI/CD tools in its life span, but never at the same time, this may involve that a developer was not happy with the CI they were using, and switched to another one. They observed 14,219 migration cases within 13,083 different repositories. The majority of them were repositories stopping the use of Travis (11K+), with the main target of these migrations being GitHub Actions (12K+). For the majority of CI/CD tools, there were more repositories stopping to use them rather than starting the use them. Only 29 out of 46,416 repositories migrated from GitHub Actions to another CI.

Finally, in the **fourth research question**, they used the Regression Discontinuity Design technique to model the effect of a specific important event by comparing the situation within a specific time window before and after the event. In their case, they used the introduction of GitHub Actions to the public as the event. Their observations showed a decrease in the growth rate in the usage of Travis, CircleCI, and Azure.

The result that surprised them the most was that GitHub Actions became very popular very quickly. About this popularity, they issued some hypotheses including the fact that GitHub Actions are completely integrated to GitHub, they

are trendy, easy to setup and use, and GitHub Actions have a large marketplace.

2.3.3 Conclusion

From the first article presented above it has been observed that, based on a sample of 3,190 active project repositories, 708 distinct Actions were used. They analyzed issues on repositories that were linked to GitHub Actions and observed that the majority of discussions were positive, meaning that developers were happy with GitHub Actions. Finally, they observed that, after the adoption of GitHub Actions on repositories, the number of commits for merged pull requests decreased, and the number of rejected pull requests increased.

From the second article, it has been observed that GitHub Actions quickly became popular after their launch, at the expense of older CI/CD tools. They observed that repositories using a CI/CD tool had a trend to change it to another one. This could be explained by the arrival of a new CI/CD tool on the market, or because the first one did not meet the developers needs.

This thesis naturally fits in this line of research. It aims to explore in greater depth some aspects of the evolving GitHub Actions ecosystem that were not yet studied. More particularly, how the marketplace is growing over time, which categories of actions are most commonly proposed on the marketplace, how fast are Actions evolving, and so on.

Chapter 3

Realization

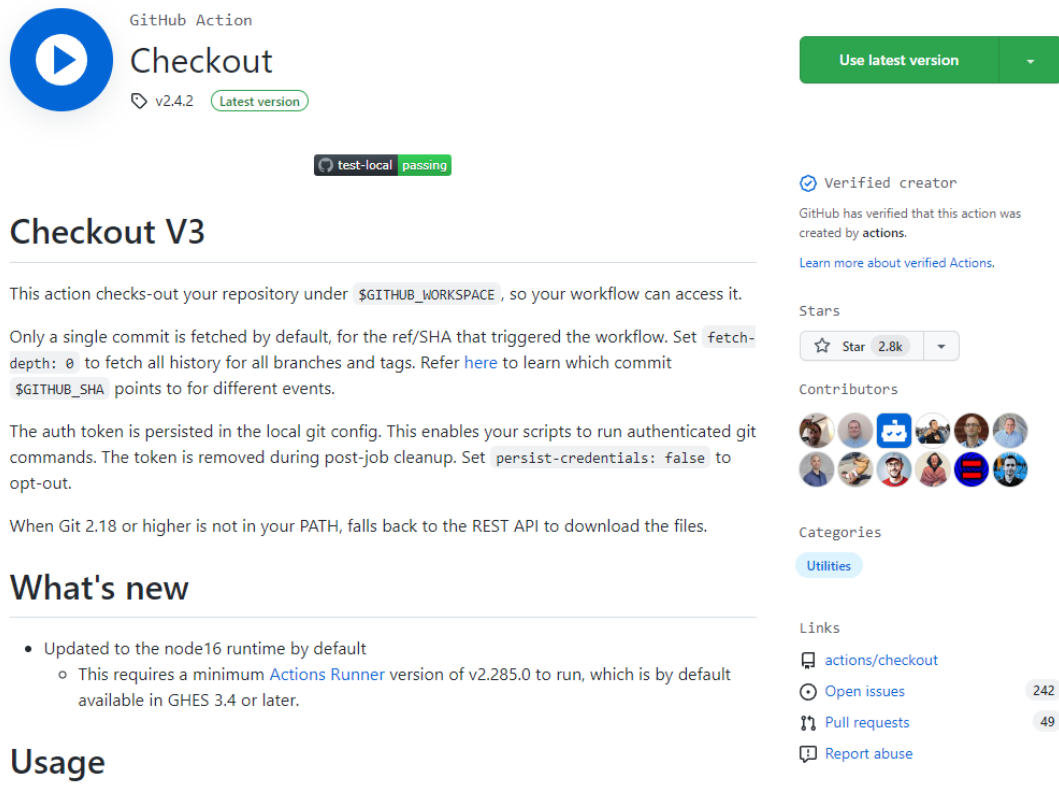
This chapter describes the technical aspects of the concretization of an empirical study. The purpose of this study is to gain a deeper understanding of how the GitHub Actions ecosystem is structured and how it evolves over time. To do this, some observations made by Golzadeh et al. [1] will be compared to new observations, and the results of new RQs will be observed. In section 3.1, the reader will find a reflection about the programming language that has been chosen for this master's thesis. In the subsequent two sections (3.2 and 3.3) an explanation is given about why and how the data has been extracted and stored. Finally, in section 3.4, insights are given about how the data has been analyzed in order to answer the different research questions discussed in chapter 4.

The source code used to extract the data and to analyze it can be found on GitHub: <https://github.com/Herostere/Master-Thesis>

3.1 Programming Language

In order to answer the different research questions of this empirical analysis, it was necessary to retrieve some data about GitHub Actions. This data can be found

on the GitHub Marketplace page of an Action (as illustrated by Figure 3.1), on its repository page (as presented on Figure 3.2), or using the GitHub REST and GraphQL API (figures 3.3 and 3.4 respectively). The GraphQL API and REST API are two distinct APIs that allow the developers to extract information found on GitHub repositories. As far as we know, there is no API to extract information about Actions on the Marketplace. The GraphQL API and REST API does not always allow the extraction of the same data. As an example, the GraphQL API do not allow the extraction of information about the contributors of a GitHub repository, while the REST API does. Furthermore, there is a difference in terms of efficiency implied by the use of these APIs, which is discussed in section 3.2.3.



Marketplace / Actions / Checkout

GitHub Action

Checkout

v2.4.2 Latest version

test-local passing

Checkout V3

This action checks-out your repository under `$GITHUB_WORKSPACE`, so your workflow can access it.

Only a single commit is fetched by default, for the ref/sha that triggered the workflow. Set `fetch-depth: 0` to fetch all history for all branches and tags. Refer [here](#) to learn which commit `$GITHUB_SHA` points to for different events.

The auth token is persisted in the local git config. This enables your scripts to run authenticated git commands. The token is removed during post-job cleanup. Set `persist-credentials: false` to opt-out.

When Git 2.18 or higher is not in your PATH, falls back to the REST API to download the files.

What's new

- Updated to the node16 runtime by default
 - This requires a minimum [Actions Runner](#) version of v2.285.0 to run, which is by default available in GHES 3.4 or later.

Usage

Use latest version

Verified creator

GitHub has verified that this action was created by [actions](#).

[Learn more about verified Actions.](#)

Stars

Star 2.8k

Contributors

Categories

Utilities

Links

- [actions/checkout](#)
- [Open issues](#) 242
- [Pull requests](#) 49
- [Report abuse](#)

Figure 3.1: Overview of the information that can be found on a GitHub Marketplace page for an Action.

Figure 3.1 gives us some information about the “checkout” Action and its owner. In this example, the owner, whose username is *actions*, is verified by GitHub. The latest version of this Action is *v2.4.2*. The repository has been starred 2.8K times, and the Action belongs to the *Utilities* category on the GitHub Marketplace. One Action can belong to a maximum of two categories: a “principal” and a “secondary” one.

Figure 3.2 gives us information about the repositories that are making use of the Action. In this example it is indicated that 3,439,422 repositories are use the “checkout” Action. Furthermore, a link to each of these repositories is available.

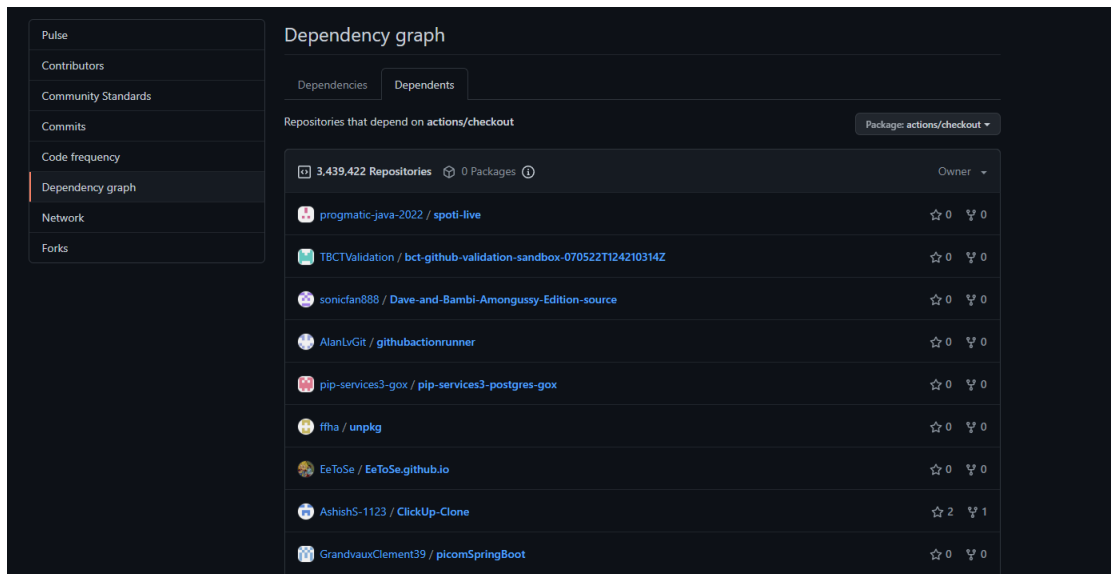
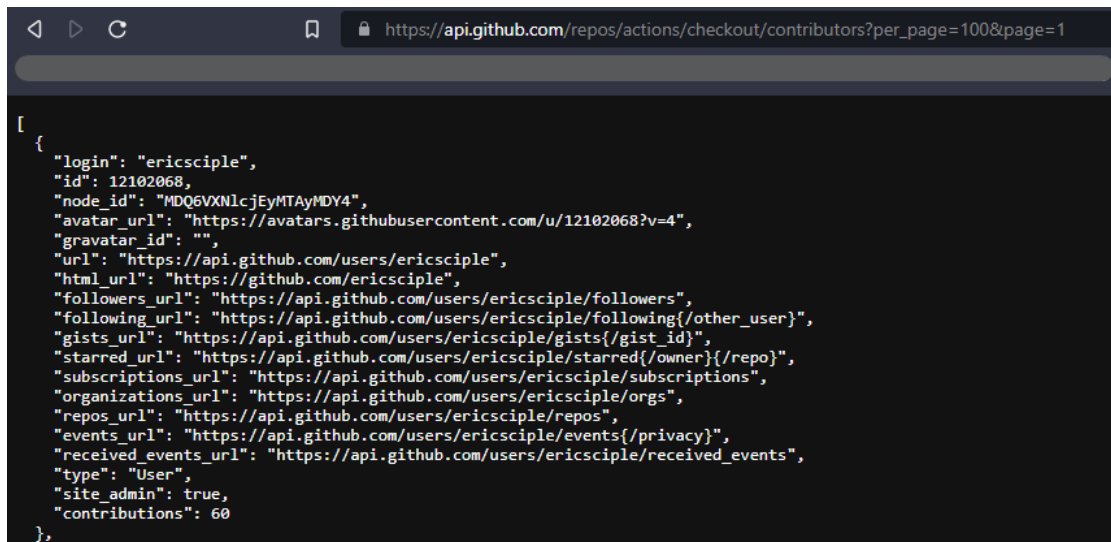


Figure 3.2: Overview of the information that can be found on a GitHub repository page for an Action.

Figure 3.3 gives an example of how the GitHub REST API can be used to find information. In this example, the API endpoint, on which the query has been issued is `https://api.github.com/repos/actions/checkout/contributors?per_page=100&page=1`, and the answer is the data visible on the web page. A request to the REST API endpoint can be issued without the use of a web browser, using a programming language, as explained in section 3.2.3. The answer to the request is formatted as a JSON file.

This answer is a list containing information about the contributors of the “checkout” Action owned by “actions”. In this example, the first element of the list is the information of the “ericsciple” contributor. Some information that can be seen on this figure are the “login” of the contributor and the “repos_url” giving the URL to the list of repositories owned by this user. In this example, the information is extracted by issuing a request to the API endpoint with a web browser.



```
[
  {
    "login": "ericsciple",
    "id": 12102068,
    "node_id": "MDQ6VXNlcjEyMTAyMDY4",
    "avatar_url": "https://avatars.githubusercontent.com/u/12102068?v=4",
    "gravatar_id": "",
    "url": "https://api.github.com/users/ericsciple",
    "html_url": "https://github.com/ericsciple",
    "followers_url": "https://api.github.com/users/ericsciple/followers",
    "following_url": "https://api.github.com/users/ericsciple/following{/other_user}",
    "gists_url": "https://api.github.com/users/ericsciple/gists{/gist_id}",
    "starred_url": "https://api.github.com/users/ericsciple/starred{/owner}/{/repo}",
    "subscriptions_url": "https://api.github.com/users/ericsciple/subscriptions",
    "organizations_url": "https://api.github.com/users/ericsciple/orgs",
    "repos_url": "https://api.github.com/users/ericsciple/repos",
    "events_url": "https://api.github.com/users/ericsciple/events{/privacy}",
    "received_events_url": "https://api.github.com/users/ericsciple/received_events",
    "type": "User",
    "site_admin": true,
    "contributions": 60
  }
]
```

Figure 3.3: Overview of the information about the contributors of an Action that can be found issuing a request to the `https://api.github.com/repos/<username>/<repository>/contributors` endpoint of the GitHub REST API.

Finally, Figure 3.4 illustrates how the GitHub GraphQL API can be used to retrieve information. In this example, the query on the left side of the figure is used to retrieve the state of the first two issues to the “checkout” Action. The response on the right side of the figure indicates that these two issues are “closed”.

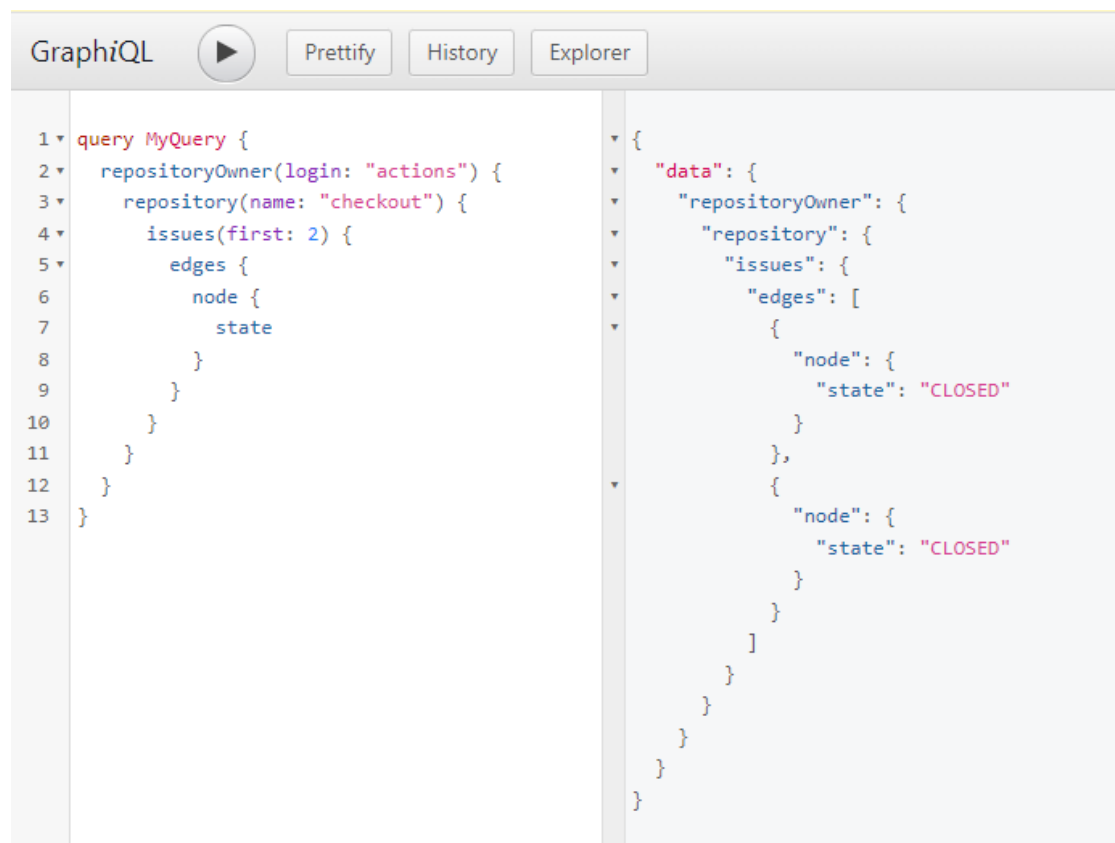


Figure 3.4: Example of information about issues of a GitHub repository extracted using the GitHub GraphQL API.

The example illustrated by the previous figure is deliberately simple for the sake of clarity, but more information about issues can be extracted by slightly modifying the query. The following example, depicted on Figure 3.5, can be used to extract the state, title, date of creation, id, name of the author, number, and URL of the first issue. More information about issues can be extracted and if the readers are looking for some more information, they are invited to consult the explorer¹ provided by GitHub.

¹<https://docs.github.com/en/graphql/overview/explorer>

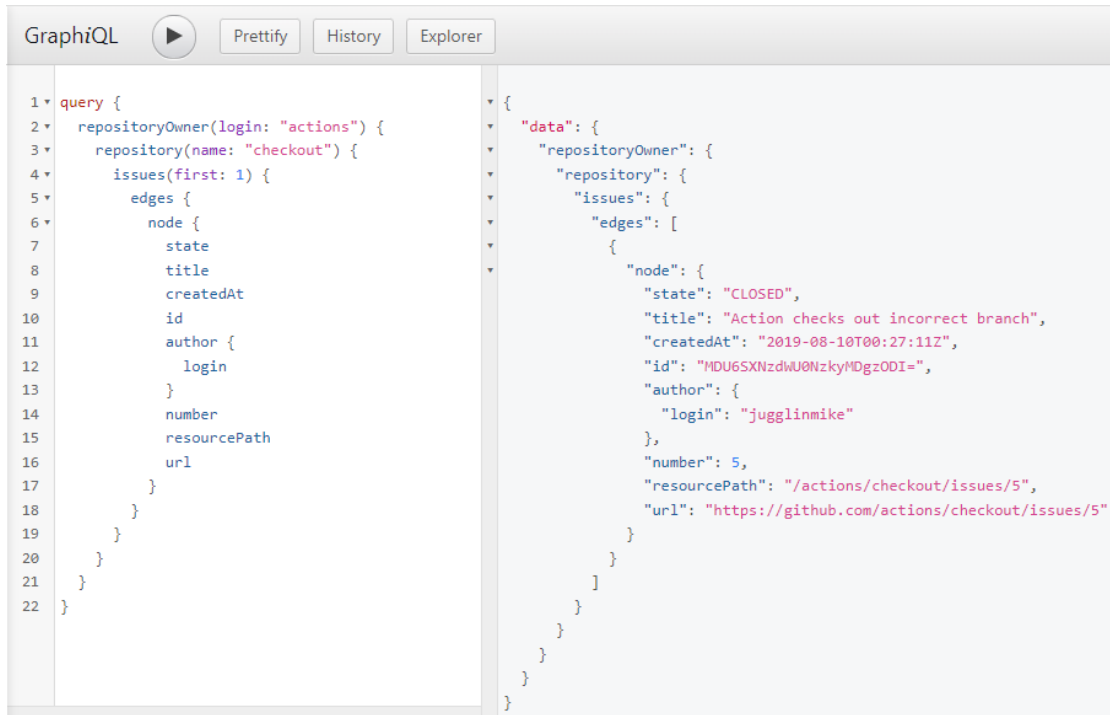


Figure 3.5: Second example of information about issues of a GitHub repository extracted using the GitHub GraphQL API.

To conduct the empirical study required for this master’s thesis, it was necessary to collect information about a large quantity (several thousands) of GitHub Actions. Such a gathering is not doable by hand. Hence, it has been chosen to automate the collecting process by using a data extraction tool developed by using a programming language.

The language that has been chosen is Python². The key reason for this choice is that Python is the main language used for data analysis. Also, Python has a large number of libraries that are useful for this work, including Matplotlib³, Seaborn⁴, BeautifulSoup⁵ and ratelimit⁶. The reasons why these libraries are useful will be

²<https://www.python.org/>

³<https://matplotlib.org/>

⁴<https://seaborn.pydata.org/>

⁵<https://beautiful-soup-4.readthedocs.io/en/latest/>

⁶<https://pypi.org/project/ratelimit/>

explained later in this chapter, in sections 3.2.2, 3.2.3 3.3, and 3.4.

3.2 Data Extraction

The first step of this master’s thesis was to gather data about GitHub Actions. This data being available in different locations; namely, the GitHub Marketplace page for an Action and its GitHub repository, it was necessary to identify which data to retrieve in which location.

To give an answer to each research question, the following data had to be retrieved for each Action:

1. the owner of the Action, as it was needed to retrieve the remaining data of this list (number 4 to 10);
2. the name of the repository of the Action, as it was needed to retrieve the remaining data of this list (number 4 to 10);
3. the name of the Action, as it was needed to retrieve to remaining data of this list (number 4 to 10);
4. the categories⁷⁸ to which the Action belongs (needed for RQ1, RQ2, RQ4, RQ5, RQ7, and RQ8);
5. if the owner is verified by GitHub, meaning that GitHub verified the ownership of the owner domain, and that the owner has confirmed its email address, and is using two-factor authentication for its organization. This

⁷To publish an Action on the Marketplace, developers have to choose a “Primary Category”, and optionally “Another Category” [14].

⁸Even if the documentation provided by GitHub [14] let us think that providing a “Primary Category” is mandatory, it appeared that some Actions on the Marketplace do not belong to a category. Therefore, this work only takes into consideration Actions belonging at least in one category.

will give insight into the trust that developers place in users who are not verified (needed for RQ7) ;

6. the list of versions of the Action and the date at which they were released (needed for RQ3);
7. the number of dependent projects⁹ (needed for RQ5);
8. the list of contributors (needed for RQ5);
9. the number of stars of the repository containing the Action (needed for RQ5);
10. the number of forks of the repository containing the Action (needed for RQ5);
11. the number of watchers of the repository containing the Action (needed for RQ5);

In addition to the items of this list, the content of YAML files in *.github/workflows/* of repositories using GitHub Actions is needed for RQ4, RQ8.

The idea to gather this data was the following: if the data can be retrieved with the GitHub APIs, the APIs are used. Otherwise, the information is scrapped from the GitHub Marketplace page or the GitHub repository of the Action. It is to be noted that GitHub did not have, at the time of this work, an API for the Marketplace that allows to obtain information about the Actions. Consulting the Marketplace page for each Action, it was possible to get their owner, their categories, the name of the Action, the name and URL of their GitHub repository, and to determine if the owner was verified by GitHub. In the repository of each Action, it was possible to find the list of releases with their tags and the date at which they were released, the number of dependent projects, the list of contributors, and the number of stars, forks and watchers.

⁹Projects using the Action in their workflows.

3.2.1 Python Selenium Library

To automate the process of collecting the data presented above, the Python Selenium¹⁰ library has been considered. Selenium is composed of a suite of tools used for the automation of actions on web browsers [15], and it to access and collect information from web pages. To do so, the developer has to specify which page they want Selenium to access, wait for Selenium to open the web browser, and wait for the page to load on the web browser. At this point, the developer has not collected any data. To overcome this, the developer has to find the relevant information on the HTML contents of the currently accessed page using XPath¹¹. The information being retrieved, Selenium will access the next page specified by the developer and the process will be the same as the one explained before (except that Selenium does not have to open a new web browser, since it is already opened). Selenium has an easy syntax, which makes it easy for a developer to use. Furthermore, its online documentation¹² is clear and up-to-date. However, Selenium may require some setup in order to be used on some operating systems, e.g. on Windows Selenium requires to download a driver and add the path to this driver in the PATH environment variable[16]. It is to be noted (cf. Selenium documentation) that the installation processes on Linux and Windows are different. Using Selenium would imply making sure that the user was using the same operating system and the same Selenium driver. Since this work was going to be done using several computers with different operating systems, another solution to automate the process of collecting the data has been considered.

¹⁰<https://www.selenium.dev/>

¹¹<https://www.w3.org/TR/1999/REC-xpath-19991116/>

¹²<https://selenium-python.readthedocs.io/>

3.2.2 Python *requests* Library

Another solution was to use the Python *requests*¹³ library. Unlike Selenium, *requests* does not need to open a web browser, wait for it to start, and wait for the web page to load. There is no need for special configuration either. The way *requests* works is as follows. The developer can specify an URL, make a GET request to the URL, and store the response as a *requests.Response* object. This object has a *text* parameter allowing the developer to inspect the HTML source code of the response. This text can be prettified using *beautifulsoup* in order to make its analysis easier. Then, an XPath expression can be used to extract the relevant information from the HTML source code, e.g. the following XPath expression is used to check if a user is verified:

```
1 xpath = '//*[@text()[contains(., "Verified creator")]]'
```

Although there are alternatives to the *requests* library, such as *urllib3*¹⁴ and *grequest*¹⁵, *requests* is the most popular one with more than 1.3M dependent projects¹⁶ on GitHub. Moreover, *requests* fulfill all the needs for this work. This is the reason why it has been chosen.

3.2.3 Extraction

Having chosen the way to retrieve the first data, here is a more precise description of the operation for the extraction of this information. First, a request is being issued to the following URL: `https://github.com/marketplace?category={category}&page={page}&type=actions`, with `{category}` and `{page}` being the

¹³<https://docs.python-requests.org/en/latest/>

¹⁴<https://urllib3.readthedocs.io/en/stable/>

¹⁵<https://github.com/spyoungtech/grequests>

¹⁶<https://github.com/psf/requests/network/dependents>

category for which to make the extraction, and the Marketplace page on which the Actions are located. It must be said that some Actions on the Marketplace do not belong to any category. Because of the way Actions are displayed to users on the Marketplace, it was not possible to make a query to extract the Actions that do not belong to any categories. Moreover, it was decided to only take Actions belonging in the “main” categories. Thus, Actions belonging to only subcategories are not taken into consideration for this work.

For this example, let’s say the category was *code quality* and the page number is 1. The URL would, therefore, look like `https://github.com/Marketplace?category=code-quality&page=1&type=actions`. Using the XPath expression

```
1  "//div[@class='d-md-flex flex-wrap mb-4']/a/@href"
```

it was possible to get a list containing the Marketplace page URL of every Action on the requested URL. Knowing the URL of the Action Marketplace page, a test has been made to check if it was a correct URL. This test consisted of issuing a GET request to the URL, and observing the response status code. A status code of 404 indicated that the URL was not accessible. If not accessible, no data for this Action can be gathered. In the other case, the response is stored and the name of the owner of the Action can be extracted from the URL. In this example, the owner was “actions”. It is the name place just after “/marketplace/”. From the response that was stored earlier, it was possible can get the link of the Action repository by using the following XPath expression

```
1  '//h5[text()="\n          Links\n          "]/following-sibling::a[1]/@href'
```

As an example, the URL would be `https://github.com/batchcorp/schema-publisher`, and the repository “schema-publisher”. Next, always on the

Marketplace page of the Action, it is possible to check if the user is verified or not using another XPath expression. In the example, the owner (*batchcorp*) was not verified. At this point, we know the name of the Action, its category on the Marketplace, the name of its owner, if the owner is verified, and the name of the repository along with its URL.

Then, it was possible to issue a request to get the data that is located in the repository. In the repository, the interesting information was the number of dependent projects. In order to get this information, a request had to be issued to the following URL: `https://github.com/{owner}/{repo_name}/network/dependents`. In our example, the URL was `https://github.com/batchcorp/schema-publisher/network/dependents`. Using another XPath expression, the number of dependents for this project could be determined. For this specific example, there was no dependent.

At this point, all the information that could not be retrieved with the GitHub APIs has been retrieved. The reason why GitHub APIs were used to get the remaining data is because it is much easier to filter data from the APIs response than to use an XPath expression. As an example, using the same Action as before, in order to get the number of contributors using the GitHub REST API a request had to be issued to the API using the following URL `https://api.github.com/repos/batchcorp/schema-publisher`. Instead of getting an HTML response, the GitHub REST API gave a response formatted as JSON¹⁷.

The GitHub REST API is not the only API provided by GitHub. Indeed, GitHub provides a GraphQL API that has also been used to retrieve information about the Actions.

Therefore, it was straightforward to extract the relevant data using the keyword associated to the wanted data. The process to gather the remaining information

¹⁷<https://www.json.org/json-en.html>

was similar. The only differences were in the URL to contact, and the keywords to use. The following table contains the relevant URLs to use with the API.

Data	URL
versions	https://api.github.com/repos/{owner}/{repo}/releases
contributors	https://api.github.com/repos/{owner}/{repo}/contributors
stars	https://api.github.com/repos/{owner}/{repo}/stargazers
watchers	https://api.github.com/repos/{owner}/{repo}/subscribers
forks	https://api.github.com/repos/{owner}/{repo}/forks
issues	https://api.github.com/repos/{owner}/{repo}/issues

Table 3.1: GitHub API URLs.

It is to be said that GitHub provides two types of API, namely REST and GraphQL [17]. To connect with the REST API, the user has to issue a request to specific URLs, as explained above. The way the GraphQL API works is slightly different. Instead of issuing requests to multiple URLs, the user only has to connect with one single URL. The entry point to this API will always be the same (of course, it is unique depending on the site for which it is configured). To get the desired data using GraphQL, the body of the request must contain a query whose parameters will vary depending on the desired data.

GraphQL APIs are an alternative to the REST APIs. With REST APIs, a developer can possibly get more information than desired. Or worse, a developer could get less information than needed, and be forced to send multiple requests to the API in order to get the same information that could be retrieved with one single GraphQL API query. It is to be noted that GitHub limits the number of hourly requests to its API to 5,000. Being forced to send multiple requests to the REST API to get the same data that could be retrieved on a single request with GraphQL API slowed down the speed at which the data were collected. Indeed,

using only the REST API the data gathering could take up to 24 hours, compared to seven hours using the GraphQL API.

To be able to retrieve all the data needed for this study, it has been necessary to send a huge number of requests (around 150k) to the GitHub website and API servers. In order to increase the speed at which requests were issued, it has been chosen to use multithreading.

Here is how it works:

1. check the number of pages for each category on the GitHub Marketplace¹⁸.
It has been observed that the maximum number of pages for a category is 50.
2. divide the number of pages by the number of threads (configurable by the user).
3. spread the pages equally in the different threads.

Each thread will retrieve the data for the Actions on the pages assigned to it. This way, the data for multiple Actions can be retrieved at the same time. It is to be noted that the use of the *ratelimit*¹⁹ Python library allowed to limit the number of requests issued to the GitHub website every minute without being concerned about the number of threads.

Some limitations have been encountered for the extraction of data. First, for each category on the Marketplace, a maximum of 50 pages of 20 Actions were available. It is unclear why GitHub does not display more Actions. As an example, if you select “show all” on the main page of the Marketplace, at the time of writing, the Marketplace announces 13K Actions but only displays $20 * 50 = 1,000$ of them. This can be seen in figure 3.6. Because of this restriction, this work only uses approximately 56% of all Marketplace Actions.

¹⁸<https://github.com/Marketplace?type=actions>

¹⁹<https://pypi.org/project/ratelimit/>

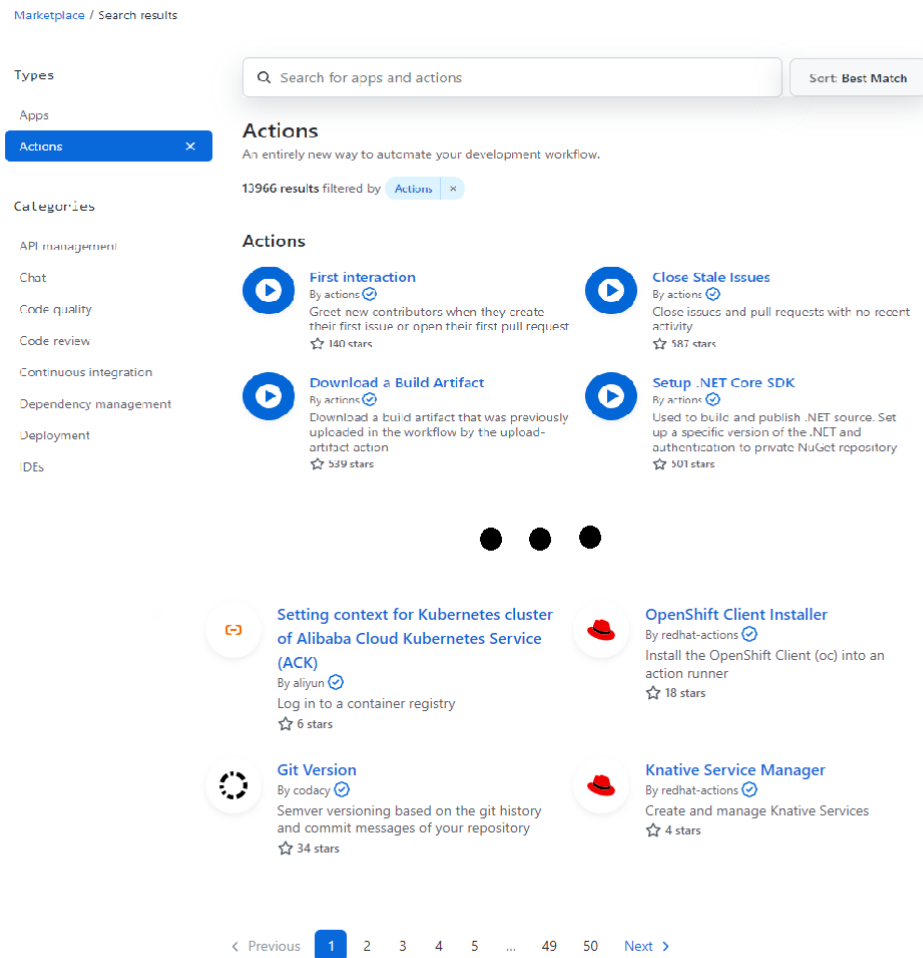


Figure 3.6: Overall number of Actions on the Marketplace.

The page has been cropped but the reader is invited to consult <https://github.com/marketplace?category=&type=actions>.
to observe this restriction.

Finally, some data were more complicated to collect. It is the case for the data needed to understand what are the most used Actions and understand how Actions used are triggered in developers workflows, because it required to get random GitHub repositories, identify if they were using GitHub Actions, and analyze the

content of their workflow files.

3.3 Data Storage

The next step was to store the data in an efficient way. By efficient we mean that the data is easy to be filtered and analyzed. Two alternatives were considered; saving the data in a text file using the JSON format and saving the data using SQLite²⁰.

An advantage to saving the data with the JSON format is the human readability of the file. An example of a JSON file containing the extracted data using the extraction tool developed for this master's thesis is presented in the following:

```
1 {
2   "RobotsAndPencils/1password-action": {
3     "category": "security",
4     "contributors": [
5       "MattKiazyk",
6       "dependabot[bot]",
7       "interstateone"
8     ],
9     "dependents": {
10      "number": 1,
11      "package_url": "https://github.com/RobotsAndPencils/
12                      1password-action/network/dependents?
13                      package_id=UGFja2FnZS0zMDQ0NDkzOTU4"
14    },
15    "forks": 10,
16    "issues": {
17      "closed": 51,
18      "open": 17
19    },
20    "owner": "RobotsAndPencils",
21    "repository": "1password-action",
22    "stars": 22,
```

²⁰<https://docs.python.org/3/library/sqlite3.html>

```

23     "verified": false,
24     "versions": {
25         "11/12/2020": "v1.1",
26         "19/11/2020": "1.0.1"
27     },
28     "watching": 4,
29     "name": "1password-secrets"
30 },
31 "ChandlerFerry/1password-action": {
32     "category": "security",
33     "contributors": [
34         "ChandlerFerry",
35         "MattKiazyk",
36         "dependabot[bot]",
37         "interstateone"
38     ],
39     "dependents": {
40         "number": 0,
41         "package_url": "https://github.com/ChandlerFerry/1password-action/
42                         network/dependents"
43     },
44     "forks": 0,
45     "issues": {
46         "closed": 1,
47         "open": 1
48     },
49     "owner": "ChandlerFerry",
50     "repository": "1password-action",
51     "stars": 0,
52     "verified": false,
53     "versions": {
54         "02/06/2021": "v1.1.6",
55         "14/05/2021": "v1.1.1",
56         "18/05/2021": "v1.1.3"
57     },
58     "watching": 0,
59     "name": "1password-secrets-forked"
60 }
61 }

```


The key for each action is the name of the owner and the name of the repository. Hence, in the above sample we can see the data related to two Actions: “1password-secrets” and “1password-secrets-forked”. We can see they are both in the same category (“security”), have a different number of versions, have different contributors, etc. The total length of the file after fetching is more than 210k lines for a weight of a little more than 6MB.

An alternative to the use of a JSON formatted text file to save the data could be to save the data in a database using the *sqlite3* Python library. The official documentation of *sqlite3* [18] gives us some insights about this library. This library allows the developer to create a database that does not require a separate server. Instead, the data is saved locally on a “.db” file. Once connected to the database, the developer can issue SQL commands to perform actions on the tables. This library is easy to use thanks to its clear syntax. The following Python code shows how to connect to a database, create a table, and insert data into it:

```
1 import sqlite3
2
3 connection = sqlite3.connect('example.db')
4 cursor = connection.cursor()
5 cursor.execute("CREATE TABLE data(id integer, name text, owner text)")
6 cursor.execute("INSERT INTO data VALUES (1,'action_name','action_owner')")
7 connection.commit()
8 connection.close()
```

Saving data in a JSON formatted text file is a good idea if the developer has to manually check the data. However, when it comes to analyzing a large amount of information, *sqlite3* allows to filter the data using one single SQL request while using the JSON formatted text file impose to parse the entire file in order to extract the desired information. This is the main reason why it was decided to use *sqlite3* to save the data. The model used for the database is represented

by Figure 3.7. The database is made of six tables: *actions*, *categories*, *version*, *contributors*, *dependents*, and *issues*. In Figure 3.7, arrows are used to indicate which private keys the foreign keys are making reference to. Private and foreign keys are indicated by “PK” and “FK”, respectively. For example, the “actions” table uses the name of the owner and the name of the repository as private key, and save information about the number of forks, stars, watchers, the name of the Action, and if the user is verified.

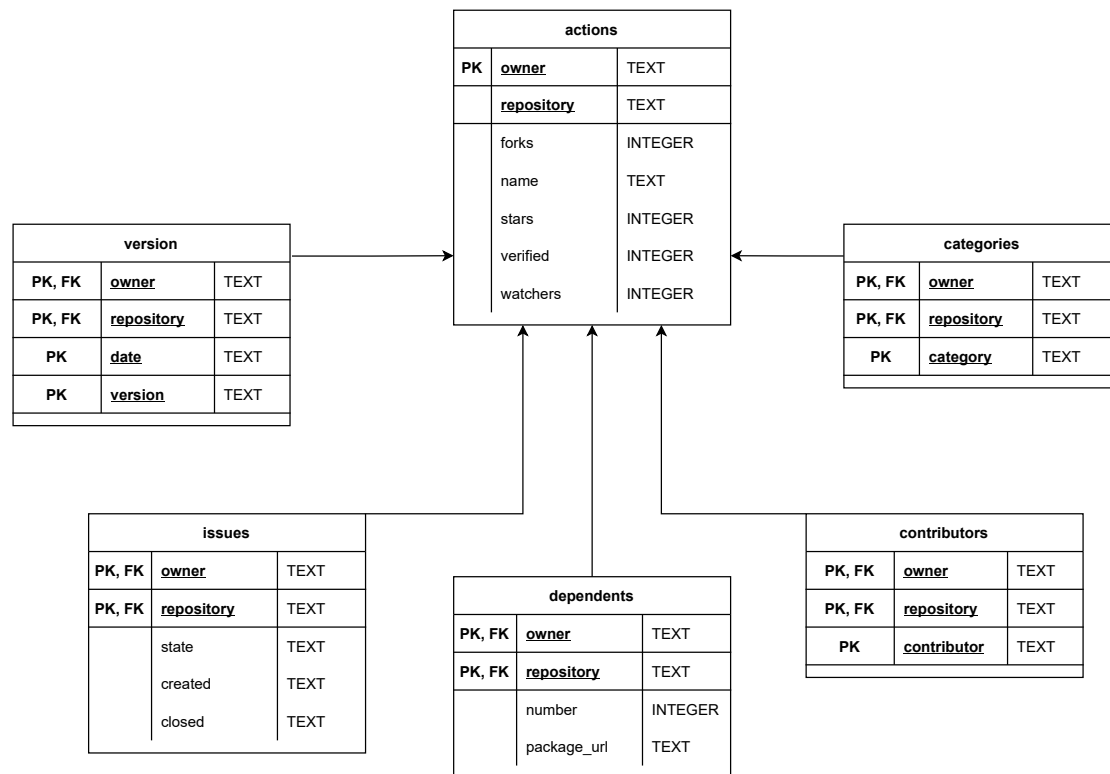


Figure 3.7: Graphical representation of the database used to save the data.

In order to make the data available before the end of the fetching execution, all the data for a category is saved to the database when the fetching for this category is completed. This way, the user does not have to wait for the execution to complete for each category before accessing the data.

3.4 Data Analysis

The third step of the work was to analyze the data previously collected. Python offers a variety of libraries that allow the visualization of data in a graphical form. In order to find the one that allows a better representation of the data, some of these libraries were compared.

The first one that was examined is *matplotlib*²¹. *Matplotlib* can be used for creating static, animated, and interactive visualizations in Python [19]. It can be used to display data using a variety of plots, including histograms, scatter plots, and polar plots. *Matplotlib* is easy to use. As an example, the following code generates a bar plot of the number of Actions per repositories.

```
1  import matplotlib.pyplot as plt
2
3  # get a dictionary with the Marketplace categories as keys,
4  # and the number of actions as values
5  actions_per_categories = number_of_actions_per_categories()
6  # make a list made of the Marketplace categories
7  categories_to_display = list(actions_per_categories.keys())
8  # make a list made of the number of Actions per categories
9  actions_to_display = list(actions_per_categories.values())
10
11 plt.bar(categories_to_display, actions_to_display, width=0.6)
12 plt.xlabel = "actions_per_categories"
13 plt.ylabel = "categories"
14 plt.xticks(rotation="vertical")
15 plt.show()
```

This code generates the plot represented in Figure 3.8.

²¹<https://matplotlib.org>

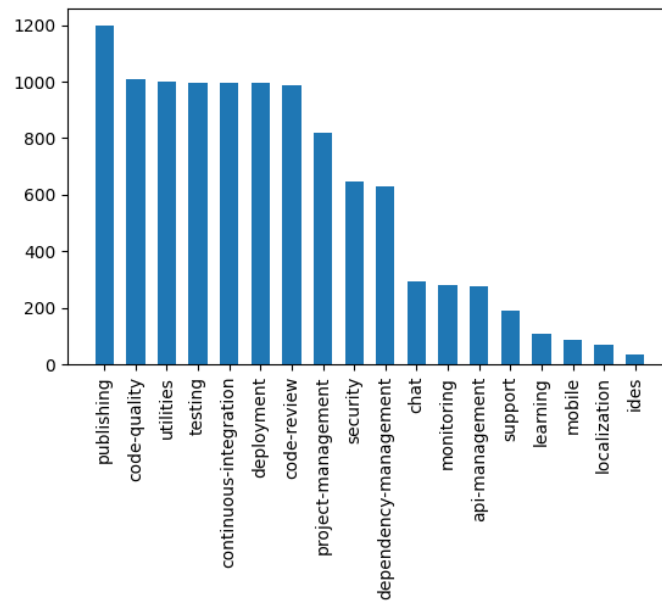


Figure 3.8: Number of Actions per categories - *matplotlib*.

Another library that was considered is *seaborn*²². *Seaborn* is based on *matplotlib* and provides a high-level interface for drawing attractive and informative statistical graphics [20]. Its syntax is simpler and more flexible than the syntax of *matplotlib*. For example, to generate a graph similar to the one presented in figure 3.8, the following Python code can be used:

```

1 import matplotlib.pyplot as plt
2 import seaborn
3
4 # get a dictionary with the Marketplace categories as keys,
5 # and the number of actions as values
6 actions_per_categories = number_of_actions_per_categories()
7 # make a list made of the Marketplace categories
8 categories_to_display = list(actions_per_categories.keys())
9 # make a list made of the number of Actions per categories

```

²²<https://seaborn.pydata.org/>

```

10 actions_to_display = list(actions_per_categories.values())
11
12 bar_plot = seaborn.barplot(
13     x=categories_to_display,
14     y=actions_to_display,
15     orient='v',
16     color="steelblue"
17 )
18 bar_plot.bar_label(bar_plot.containers[0])
19 plt.xticks(rotation='vertical')
20 plt.show()

```

This code generates the plot presented in Figure 3.9.

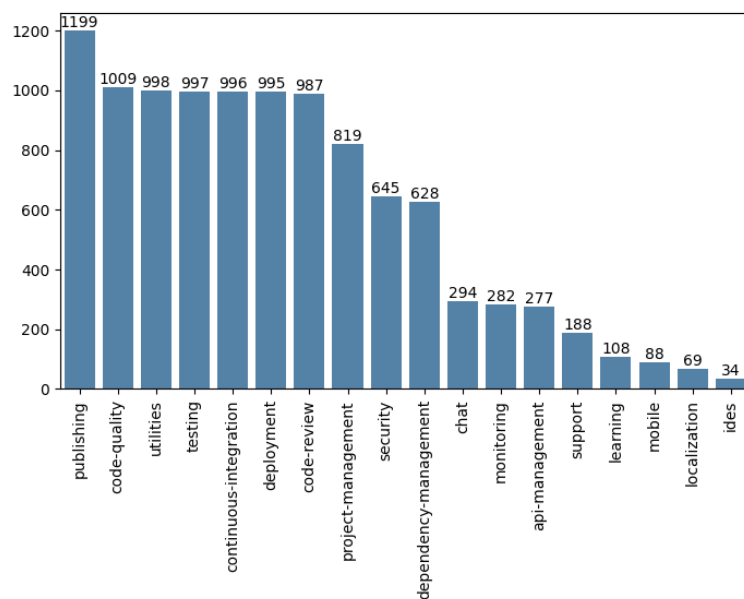


Figure 3.9: Number of Actions per categories - *seaborn*.

Checking the activity history on the GitHub repositories of both *matplotlib*²³ and *seaborn*²⁴, it has been noticed that both of them were well maintained. For

²³<https://github.com/matplotlib/matplotlib>

²⁴<https://github.com/mwaskom/seaborn>

example, at the time of writing, the last “commit” in the *matplotlib* repository has been made two hours ago, and 17 hours ago on the *seaborn* repositories. Also, the frequency of “commits” for both libraries was high, with hundreds of commits every month. Because the syntax of *seaborn* was found to be clearer than the syntax of *matplotlib*, it is the one that was selected to visually analyze the data for this work.

At this point, all the needed was collected using Python, the Marketplace pages of the Actions, their repositories, and the GitHub APIs. The data has been saved in a database, and a tool to visually analyze it was chosen.

In order to make statistical observations (e.g. median, quartiles, maximum, minimum, IQR) about a set of values, it was chosen to use Python libraries. The first one is the *statistics*²⁵ Python library. The other libraries are *NumPy*²⁶ and *SciPy*²⁷. Since their usage is quite simple (import the library and call the desired methods) the reader is invited to check their documentation. However, a list of the methods that were useful for this master’s thesis is given in Table 3.2.

Library	Method
scipy.stats	mannwhitneyu
numpy	percentile
statistics	median

Table 3.2: Useful methods for the statistic analysis.

The *mannwhitneyu* method was used to perform a Mann-Whitney U test in RQ3, and the *percentile* (resp. *median*) methods were used to compute the first and third percentiles (resp. the median) of a list of numbers in multiple RQs.

²⁵<https://docs.python.org/3/library/statistics.html>

²⁶<https://numpy.org/doc/>

²⁷<https://scipy.org/>

Chapter 4

Results

This chapter reports the results of the study, with each section representing a specific research question.

4.1 RQ1: How Fast Does the GitHub Marketplace Grows Over Time?

This RQ aims to understand how fast the GitHub Marketplace grows over time. A growth could mean that the GitHub Actions are still gaining in popularity, while a decrease could indicate that the GitHub Actions are losing in popularity. Since Golzadeh et al. [1] observed a growth in popularity for GitHub Actions, this question aims to confirm their observations. The data used to answer this question were gathered from July 3, 2022, to July 31, 2022. It was not possible to collect data on a longer time period because the first version of the program (made between January 2022 and June 2022) did not correctly collect the data needed to make the observation; and the new corrected version has been functional starting from July 3, 2022. Because of the time constraints imposed for this master's thesis, a longer period of time was not available to collect information about GitHub

Actions.

We analyzed the growth of the overall number of Actions on the GitHub Marketplace. Since Actions can fall in a maximum of two categories, Actions were only counted once in the total number of Actions, implying that the sum of Actions in all categories was higher than the overall number of Actions. Using the program developed for this master’s thesis, it was possible to gather 7,973 Actions from the Marketplace on July 3, 2022. On July 31, 2022, the number of Actions gathered from the Marketplace using the same program was up to 8,114. Within a month, at least 141¹ Actions were added to the GitHub Marketplace. The growth of the overall number of Actions on the Marketplace is illustrated by Figure 4.1, and the number of Actions gathered for each week are listed in Table 4.1. Furthermore, the categories with the most new Actions added between July 3, 2022, and July 31, 2022, were “code quality” to which 107 Actions were added, “security” with 24 Actions added, and “project management” with 20 Actions added. This might indicate that developers have interest in Actions belonging to this categories.

It has been observed that this growth is characterized by the number of new Actions added to the Marketplace, and the number of Actions removed from the Marketplace, which is illustrated by Figure 4.2. This figure depicts the number of newly added Actions (resp. Actions removed) since the previous time stamp. Hence, the first green (resp. orange) point in the plot represents the number of new Actions (resp. Actions removed) since July 3, 2022. To identify the number of new Actions added to the Marketplace, the list of Actions gathered in each timestamp has been compared to the list of Actions of the previous timestamp. When an Action is in the list of the most recent timestamp but not in the list of the previous timestamp, this means that the Action has been added to the Marketplace. In the opposite case, i.e. when the Actions were in the previous timestamp but not

¹This number might be higher depending on the data about Actions that could not technically be gathered on the Marketplace.

in the most recent timestamp, this means that the Actions were removed from the Marketplace. The biggest weekly growth has been observed between July 24, 2022, and July 31, 2022, with a growth of 100 Actions. This observation suggests that the number of Actions on the Marketplace grows over time, but a study over a longer period would reinforce this observation.

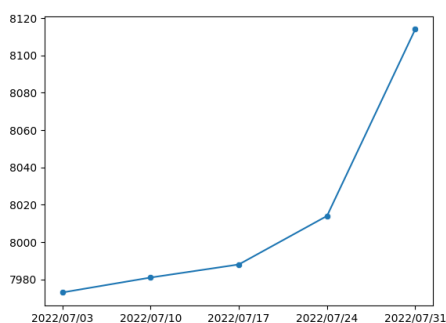


Figure 4.1: Weekly number of Actions on the Marketplace.

Dates	# of Actions
2022/07/03	7,973
2022/07/10	7,981
2022/07/17	7,988
2022/07/24	8,014
2022/07/31	8,114

Table 4.1: Weekly number of Actions on the Marketplace.

Even if the overall number of Actions grew every week, two categories had fewer Actions in the last data gathering compared to the first gathering. This is the case for the “ides” and “learning” categories. Both categories had a loss of one Action between July 3, 2022, and July 31, 2022. This loss is illustrated by Figure 4.3 for the “ides” category, and by Figure 4.4 for the “learning” category. There are two explanations to this loss. Either Actions were deleted from the Marketplace by their owners, or the owners have changed the categories to which the Actions belong. An Action can belong to a maximum of two categories, and it has been observed that some Actions are sometimes removed from a category or assigned to a new category. Table 4.2 lists the number of times Actions had a new category assigned and the number of times Actions got deleted from a category. One can observe that the category that is the most assigned to or deleted from already existing Actions is “code quality”. Overall, a new category was assigned

115 times to Actions already on the Marketplace, and a category was removed from Actions already on the Marketplace 42 times. This observation implies that when some Actions are removed from a category, it is more likely that they have been removed from the Marketplace by their owner, instead of having been removed from a category (but still remaining in another category). Finally, an in-depth observation suggested that Actions were never totally moved from their original categories. Indeed, Actions belonging to one category were never removed from this category. A new one was sometimes added. On the other hand, Actions belonging to two categories were sometimes removed from a category and a new category was sometimes added later, so one of the two original categories never changed.

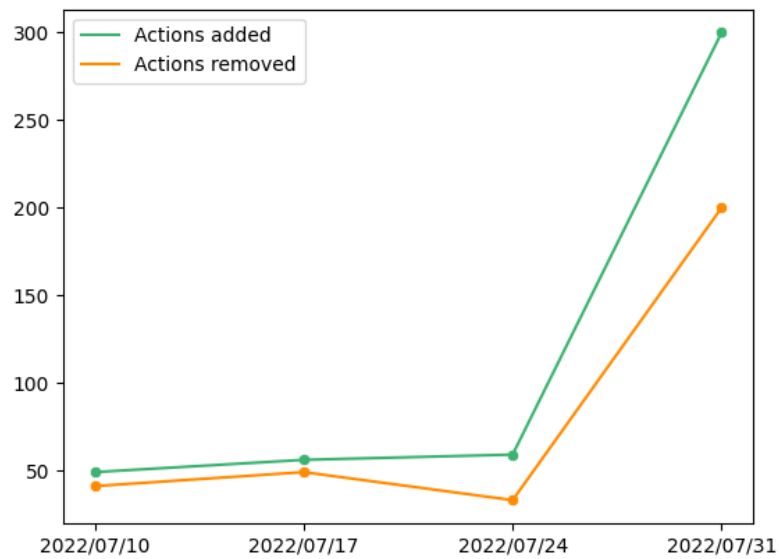


Figure 4.2: Number of new Actions added to and removed from the Marketplace.

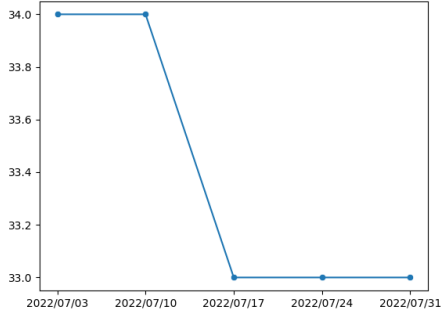


Figure 4.3: Weekly number of Actions in the “ides” category.

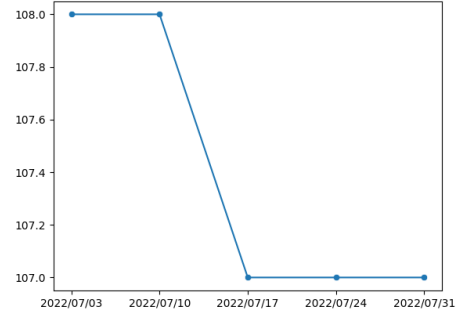


Figure 4.4: Weekly number of Actions in the “learning” category.

Categories	Times added	Times deleted
Code quality	80	13
Publishing	18	17
Testing	7	4
Continuous integration	4	1
Utilities	4	2
Deployment	1	0
Code review	1	4
Learning	0	1

Table 4.2: Number of times an Action is added to and removed from a category.

At least 141 Actions were added on the Marketplace within a month. However, even if the overall number of Actions on the Marketplace was growing, some categories (“ides” and “learning”) experienced a decrease in their number of Actions. Moreover, the categories “code quality”, “security”, and “project management” were identified as the categories with the most new Actions.

The majority of lost Actions for a category was due to users deleting Ac-

tions they owned on the Marketplace, and not due to them removing the Actions from the category. Furthermore, it has been observed that Actions were never “moved” from their original categories; either they were removed from a category, or added to a category, but never changed from their original categories.

4.2 RQ2: Which Categories of Actions Are Most Commonly Proposed on the Marketplace?

This question aims to identify the categories of Actions on the Marketplace that developers might be mostly contributing to. A category with a large number of Actions might indicate a need for developers to have a variety of Actions from this category, while a small number of Actions might indicate that developers do not need a variety of Actions from this category. The data collected on July 31, 2022 has been used to answer this question.

To answer this question, a first observation, represented by Figure 4.5, has been made about the number of Actions belonging to each category.

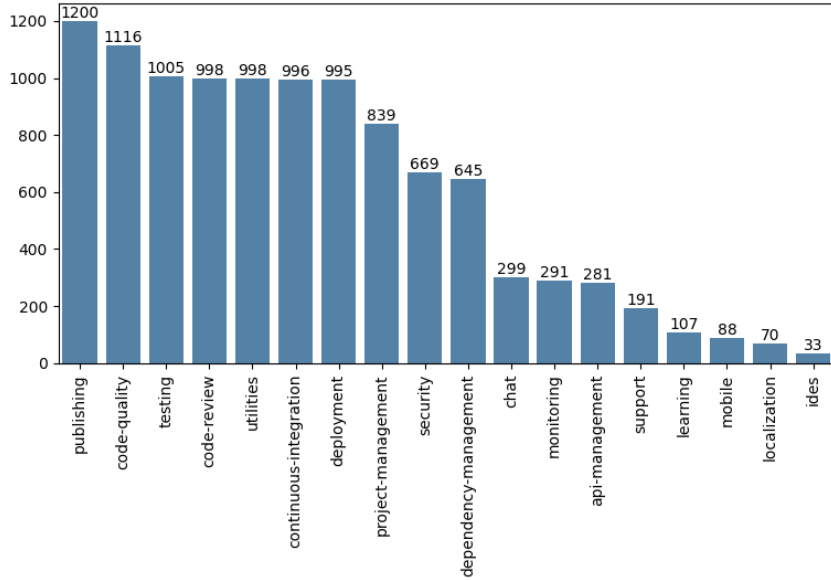


Figure 4.5: Number of Actions by category on the Marketplace on July 31, 2022.

The seven most represented categories, i.e. the categories with considerably more Actions than the others, are listed below. A description of the most starred Action referring to GitHub is given:

Publishing: Those Actions are designed to automatically publish the code. The most starred Action is *continuous-reforestation*². Its purpose is to automatically plant a tree when the Action is triggered. Trees are planted using the Reforestation as a Service provided by *DigitalHuman*³.

Code quality: Those Actions are used for code reviewing, including code quality, security, tests coverage, etc. They analyze the code submitted through pull requests [3]. The most starred Action is *super-linter*⁴. Its main purpose is to prevent broken code from being uploaded to the default branch of a repository.

Testing: Actions that are used to increase code auditing. Those Actions are an-

²<https://github.com/marketplace/actions/continuous-reforestation>

³<https://digitalhumani.com/>

⁴<https://github.com/marketplace/actions/super-linter>

alyzing for bugs in code. The most starred Action is *Is Website BVulnerable*⁵. Its purpose is to find publicly known security vulnerabilities in a website’s frontend JavaScript libraries.

Code review: Actions to ensure that the code meets quality standards. The most starred Action is the same as for code quality, i.e. *super-linter*.

Utilities: Actions that automate diverse steps of the development workflow on the GitHub platform. These Actions are often used in support of other Actions [3]. The most starred Action is *Metrics embed*⁶. Its purpose is to generate metrics that can be embedded everywhere.

Continuous integration: Actions that build and test code when it is pushed on GitHub. The most starred Action is *TruffleHog OSS*⁷ which purpose is to find leaking credentials used in a repository.

Deployment: Actions responsible for building and deploying applications. The most starred Action is *yq - portable yaml processor*⁸, whose purpose is to provide a lightweight and portable command-line YAML, JSON and XML processor.

Kinsman et al. [3] determined, among the 708 distinct Actions they observed, that the most represented categories were: “continuous integration”, “utilities”, “deployment”, “publishing”, “code quality”, “code review”, and “testing”. Our own analysis confirms that these categories were also the most represented on the Marketplace on July 31, 2022, although in a slightly different order. This could give an indication about the categories of Actions that might be the most interesting for developers. These categories include 67.53% of the overall number of Actions. It is to be noted that the overall number of Actions in the categories is 10,821, which is bigger than the number of Actions presented in RQ1. This

⁵<https://github.com/marketplace/actions/is-website-vulnerable>

⁶<https://github.com/marketplace/actions/metrics-embed>

⁷<https://github.com/marketplace/actions/trufflehog-oss>

⁸<https://github.com/marketplace/actions/yq-portable-yaml-processor>

is explained by the possibility for an Action to have a primary category and an optional secondary category.

Figure 4.6 presents a box plot summarizing the distribution of the number of Actions per category. In this figure one can observe that the maximum number of Actions for a category is 1,200 and the minimum is 33. This represents the number of Actions respectively belonging to the *publishing* and *ides* categories. Still based on this figure, the first quartile is 213.5, the third quartile is 997.5, and the median is 657. Based on these values, the interquartile range is 784, and no outliers have been observed. These values are listed in Table 4.3.

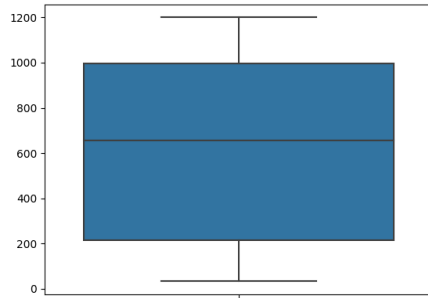


Figure 4.6: Box plot for the distribution of Actions per category.

Median	636.5
Q1	210.25
Q3	995.75
IQR	785.5
Maximum	1,199
Minimum	34

Table 4.3: Values for the number of Action for each category on July 3, 2022.

Seven categories out of 18 had considerably more Actions than the other ones, and they represented 67.53% of Actions in the categories. This might indicate that developers were more interested by Actions belonging to the *publishing*, *code quality*, *testing*, *code review*, *utilities*, *continuous integration* and *deployment* categories. These categories were the same that Kinsman et al. [3] observed as being the most represented ones.

4.3 RQ3: How Fast Are Actions Evolving?

The purpose of this question is to determine how fast and how often Actions are getting new releases. High rate of new releases might suggest that Actions are evolving fast and getting improvements recurrently. On the other hand, a low rate of new releases might suggest that Actions are not evolving so fast, and not being improved. To do so, the dates and releases tags of Actions gathered on July 31, 2022 were analyzed. The dates being stored in the following format “2022-05-25 02:56:09”, it was necessary to compute the number of days, hours, minutes and seconds between two dates using the Python *datetime*⁹ library. The following Python code shows how it works:

```
1 from datetime import datetime
2
3
4 first_time = datetime.strptime("2022-05-25 02:56:09", "%Y-%m-%d %H:%M:%S")
5 second_time = datetime.strptime("2022-07-24 12:32:32", "%Y-%m-%d %H:%M:%S")
6 difference = second_time - first_time
7 elapsed_seconds = difference.seconds
8 elapsed_days_expressed_in_seconds = difference.days * 86400
9 seconds_between_times = elapsed_seconds + elapsed_days_expressed_in_seconds
```

The output of this example is 5,218,583, and represents the number of seconds between the two dates. It is trivial to compute the number of days, minutes, and seconds represented by this value. Here, the number of days is 60, the number of hours is 9, the number of minutes is 36, and the number of seconds is 23. This computation has been made to identify the number of seconds between two releases of the same type, i.e. two major releases, two minor releases, or two patches. Major releases for Actions are important changes in the Actions, including: improvements in the source code (e.g. more efficient code that is faster to execute), the addition

⁹<https://docs.python.org/3/library/datetime.html>

(resp. removal) of new (reps. outdated) features, etc. Minor releases for Actions reflect less important changes in the Actions, including: the addition of a new limited feature, and small updates of existing features. Finally, patch releases for Actions are used to fix bugs and make minor improvements without adding new functionalities.

The 42,180 release tags found in the repositories analyzed for this RQ have been used to identify if a release was a major, minor, or a patch release. Most of the time, release tags follows the “v<major release number>.<minor release number>.<patch number>” semantic. Since developers are not constrained to follow the previous semantic for the tags of their releases on GitHub, and since it was necessary to make some choices on the way to identify if the releases where major, minor, or a patch, the choice has been made to ignore releases without tag and releases with a tag not containing any number. There were 471 out of 42,180 ($\sim 1.12\%$) ignored releases, which is low enough to not impact the observations. The reason for this choice is that the two previous scenarios do not allow an identification of the release type. This way, releases using “main”, “master”, “latest”, etc., as tag were not considered to answer this RQ, but all other tags were. To identify the release number, the Python *packaging*¹⁰ library has been used. This library helps to identify version numbers that follow the Python Enhancement Proposal (PEP) 440¹¹. Since developers are not constrained to follow this version name formatting, when a release tag not following PEP440 is parsed using *packaging* it creates a *LegacyVersion* object. When such an object is detected, a regular expression is used to extract what appeared to be a release number. The following code sample shows how releases types have been identified:

¹⁰<https://packaging.pypa.io/en/latest/>

¹¹<https://peps.python.org/pep-0440/>

```

1 from packaging import version as packaging_version
2
3 import re
4
5
6 version = packaging_version.parse("v.1.2.4")
7 test_version = re.search(r"\d+(\.\d+){0,2}", version)
8 if type(version) is packaging.version.LegacyVersion and test_version:
9     test_version = test_version.span()
10    string_version = version[test_version[0]:test_version[1]]
11    version = packaging_version.parse(string_version)
12
13 major_number = version.major
14 minor_number = version.minor
15 patch_number = version.micro

```

This way of identifying the type of release is not flawless. Imagine the tag being “dev-238askjgeeslkfsadf39”. The above algorithm will determine 238 as the major release number, 0 as the minor release number, and 0 as the patch number. However, rejecting any tag that does not follow the PEP440 would not work either because tags like “v.0.1” do not follow PEP440 (due to the dot between the “v” and the “0”). It clearly indicates that the major release number is 0, the minor release number is 1, and the patch number is 0. Overall, 672 ($\sim 1.6\%$ of the 42,180 tags analyzed) tags were identified as legacy and converted. Table 4.4 gives some examples in the form of regular expressions of accepted release tags formats.

Regular expressions	Examples
<code>v\d+.*</code>	<code>v0.0.1</code>
<code>\d+.*</code>	<code>0.1</code>
<code>v.\d+.*</code>	<code>v.0.2.384</code>
<code>release\d+.*</code>	<code>release1.3.3</code>
<code>v\d+.*-alpha</code>	<code>v1-alpha</code>

Table 4.4: Non-exhaustive list of examples of accepted release tag formats.

Another problem that has been encountered is an inconsistency in the release numbers, e.g. the earlier release tag was *v1.0* and at a later date the release tag was *v0.1*. When the comparison between two tags is carried out, if the version number represented by the tag of the oldest one is higher than the version number represented by the tag of the most recent one, the time between those two releases is not considered, and the next comparison is based on the version number of the most recent one. As an example, here is a list of dates with their associated release tag: `[('2022-05-24 03:32:25', 'v1'), ('2022-05-25 02:56:09', 'v2.1'), ('2022-05-25 17:10:56', 'v1.2'), ('2022-06-01 23:44:34', 'v1.3')]`. The reader can note the inconsistency between the second and third elements of the list. The algorithm used to analyze this list compared the first two elements of the list and noted that a major release had been made. Then, while making the comparison between the second and third element, it detected that there was a downgrade in the release tag. This comparison was therefore not considered, and the next comparison was made between the third and fourth elements of the list. Furthermore, a manual observation revealed that the naming convention used for tags sometimes evolved over time, e.g. the owner “bump-sh” uses “0.1” as first format, then uses “v1.0.0” as format. This evolution in the format used in the release tags is not a problem for this analysis since the algorithm used can extract version numbers from a variety

of formats.

Finally, it is to be noted that some repositories of Actions had their oldest release tag indicating *v6.0.0* or alike, indicating that some older releases might have been deleted by the developers. An explanation might be that these older releases never actually got published on the Marketplace. This particularity, along with the way to identify the version numbers and the inconsistencies in release tags, may suggest the existence of a threat to validity in the results of this question.

Overall, 43,748 releases were identified within the collected data. Among these releases, 729 release tags were identified as *LegacyVersion* and converted as explained above. 536 release tags were refused (because of the absence of tag or number in the release tag), and 1,957 inconsistencies were detected. The number of refused tags and inconsistencies represents 5.7% of the overall number of releases, which might indicate a small bias in the following observations.

Table 4.6 summarises the number of updates that were carried out by type of release. Based on these values, we hypothesize that Actions receive more patches than minor updates, and less major updates than minor or patch releases. To ensure the veracity of this hypothesis a Mann-Whitney U test has been performed. A description of this statistical test provided by Corder et al. [21], states that this test is used to compare two independent samples. The way this test works is explained as follows

“The two samples are combined and rank ordered together. The strategy is to determine if the values from the two samples are randomly mixed in the rank ordering or if they are clustered at opposite ends when combined. A random rank ordered would mean that the two samples are not different, while a cluster of one sample’s values would indicate a difference between them.”

Before performing the Mann-Whitney U test on the sample of seconds for the

major updates, minor updates, and patch updates, we have to make sure that these samples are independent. To do so, a t-test was performed. Peter C. Bruce [22] states that this test is an hypothesis test, and can be used to determine if two samples could come from the same population, i.e. the two samples are dependents. We are using the null hypothesis H_0 “the two samples come from the same population”, and the alternative hypothesis H_a “the two samples come from two different populations”. The Python library *scipy* was used to perform this test, and the results are listed in Table 4.5.

Comparison	p-values	p-values <0.01
Major / minor	7.5370e-29	True
Major / patch	2.4236e-221	True
Minor / patch	1.5419e-139	True

Table 4.5: t-test: p-values.

Because all p-values are smaller than the critical value of 1%, we have to reject the null hypothesis, i.e. the two samples come from two different populations. Having this information, a Mann-Whitney U test can be performed.

Using the Mann-Whitney U test, with the null hypothesis “the two populations are not statically different” and alternative hypothesis “the two populations are statically different”, a comparison has been made between the sample of seconds for the major updates, minor updates, and patch updates. Given a confidence level of 0.01, the comparison between the major and minor updates has given a p-value of 0.00000145126 which is smaller than 0.01. Hence, the null hypothesis is rejected, meaning that major updates occur less frequently than minor updates. The same goes for the comparison between major and patches, and minor and patches. The p-values for these comparisons are listed in Table 4.7.

Types	Number of updates	Comparison	p-values	p-value <0.01
Major	3,527	Major / minor	2.6745e-05	True
Minor	10,277	Major / patch	1.2535e-39	True
Patch	19,608	Minor / patch	2.4280e-261	True

Table 4.6: Number of updates by type of release.

Table 4.7: Mann-Whitney U test: p-values.

This is a result that could be expected since patch releases are small fixes of the software, minor releases are fixes along with the addition of software functions, and major updates are improvements of the whole software.

Knowing the results of the Mann-Whitney U test performed above, the time between versions of the same type has been compared. It has been observed that, in addition to being the type of release that arises more frequently than the other types of releases, patches are also the ones that take the least amount of time to switch between, e.g. switching from version *1.0.0* to version *1.0.1*. Indeed, values in Table 4.8 show that the median for the time to switch between a new patch release is 186,207.5 seconds, which is lower than the median for minor releases (1,242,152 seconds) and major releases (790,733 seconds). However, it has been observed that major versions were released faster than minor versions. Effectively, major versions were released with a median of 1,242,152 seconds, and minor versions were released with a median of 790,733 seconds, even though major versions were released less frequently than minor versions. These numbers are reported in Table 4.8, and Figure 4.7 represents these values through box plots. Because it was observed that the data distribution represented on these box plots were hardly skewed, the “y axis” has been scaled logarithmically.

	Major	Minor	Patch
Minimum	0	11	3
Q1	2,300	609,60	2,161
Median	790,733	1,242,152	186,207.5
Q3	11,669,124	6,826,207	2,190,874.5
IQR	11,666,824	6,765,247	2,188,713.5
Maximum	107,522,989	107,934,785	107,525,569

Table 4.8: Seconds between major releases.

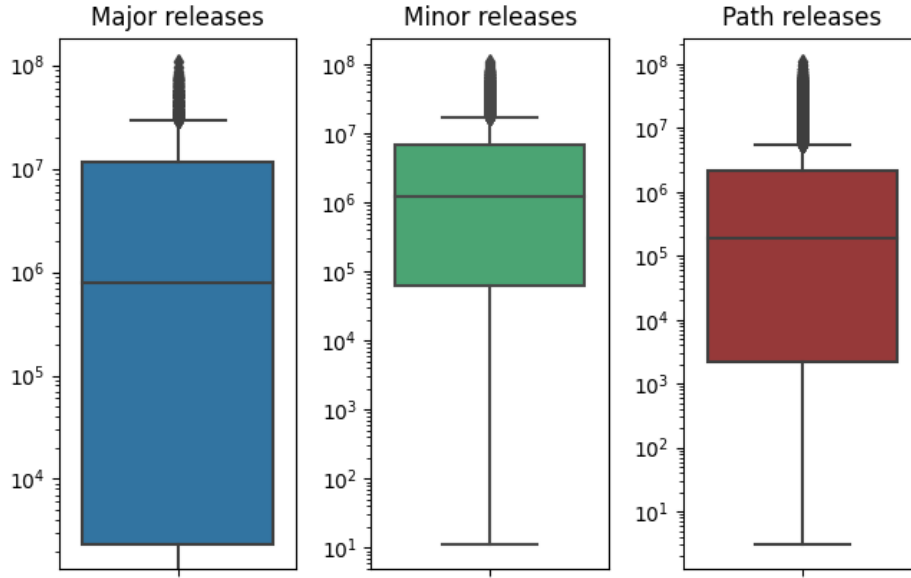
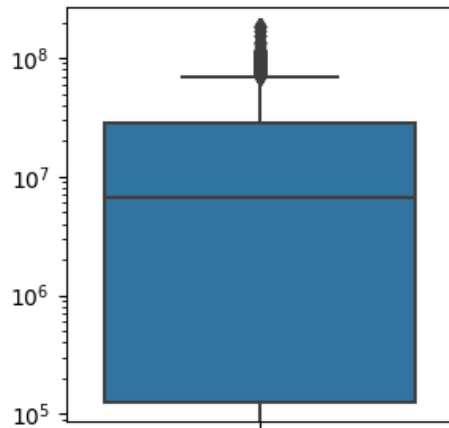


Figure 4.7: Distribution of the time between major, minor, and patch releases, respectively.

Finally, the time between releases of any types has been observed. The median to switch between releases is 6,706,637 seconds, which is equivalent to 77 days, 14 hours, 57 minutes, 17 seconds. Moreover, the third quartile suggests that 75%

of Actions got a new release within 326 days, 2 hours, 28 minutes, 47 seconds (28,175,327 seconds). Table 4.9 lists the values characterizing the distribution of the time for an Action to get a new release of any type, and Figure 4.8 represents the distribution of seconds between releases of any types.



Minimum	0
Q1	125,796.5
Median	6,706,637
Q3	28,175,327.0
IQR	28,049,530.5
Maximum	188,164,965

Table 4.9: Seconds between releases.

Figure 4.8: Distribution of the time between releases.

One can observe from the above values that 25% of the new releases are made in less than 1.0 day, 10.0 hours, 56.0 minutes, 36.5 seconds (125,796.5 seconds), and 50% of releases are made in less than 77 days, 14 hours, 57 minutes, 17 seconds (6,706,637). These values suggest that the majority of releases arrive slowly. One explanation for this observation could be the proportion of Actions being not, or poorly, maintained. A closer look at the time between releases of maintained Actions could give more accurate information about the time between releases.

Major releases occur less frequently than minor and patch releases, and patch releases occur more frequently than minor and major releases. In addition to being the most frequent type of release, patches are also the type of release

that take the least time to switch from one version to another, followed by major releases. Minor versions take a slightly longer time to switch from one version to another.

Another observation showed that releases are coming slowly. However, we suggest to the next studies to take a closer look at the time between releases for maintained Actions, instead of the overall Actions.

4.4 RQ4: Do Workflows Use Multiple Actions?

This question aims to understand if GitHub repositories using GitHub Actions are using Actions available on the Marketplace in their workflow files, and aims to get an idea of the number of Actions they are using per workflow files. The observations of this question will give a first idea about the most popular Actions available on the Marketplace, along with the number of Actions that are used in workflow files of repositories making use of Actions.

In order to answer this RQ, it was necessary to collect a large number of workflow files from GitHub repositories that are using GitHub Actions. The criteria to select a workflow file are: the repository must use GitHub Actions and must have been updated in the last twelve months. In order to check if the repository is using GitHub Actions, the presence of files with the *.yml* in the *.github/workflows* is checked. The fact that the repository must have been updated in the last twelve months indicates that the repository is not discontinued (or at least this has not been the case for a long time) and is more likely to use recent GitHub Actions. Indeed, new Actions are launched every day on the Marketplace. Moreover, this ensures that the repository has been updated after the launch of GitHub Actions on November 2019. The GraphQL API of GitHub has been used to collect the workflow files. A first query was issued to get a list of the public repositories available on GitHub that have been updated in the last twelve months. For each

repository found with the first query, a second query has been issued to get the content of their *.github/workflows* folder. If this folder did not exist, the repository was ignored. Otherwise, the contents of the workflow files were analyzed to extract the Actions available on the Marketplace that have been identified as being used.

The following observations are based on a set of 50,000 workflow files selected randomly, originating from public repositories available on GitHub. These repositories were identified as using GitHub Actions by the presence of files with the *.yml* extension in their *.github/workflows/* folder. Analyzing this set, it has been observed that 25% of the workflow files are using less than one Action. Moreover, the median (resp. third quartile) showed that workflow files are using less than two (resp. three) Actions. Finally, the minimum number of Actions on a workflow file is zero, and the maximum is 19. These results are pictured in Table 4.10, and represented as a box plot in Figure 4.9. Furthermore, 1,212 workflow files have been identified as not using Actions available on the Marketplace. This represents 2.4% of the workflow files. However, if we decide to ignore the “checkout” Action from this observation (which is used in 43,762 out of 50,000 workflow files), the number of workflow files not using Actions available on the Marketplace increases to 10,597, which represents 21% of workflow files.

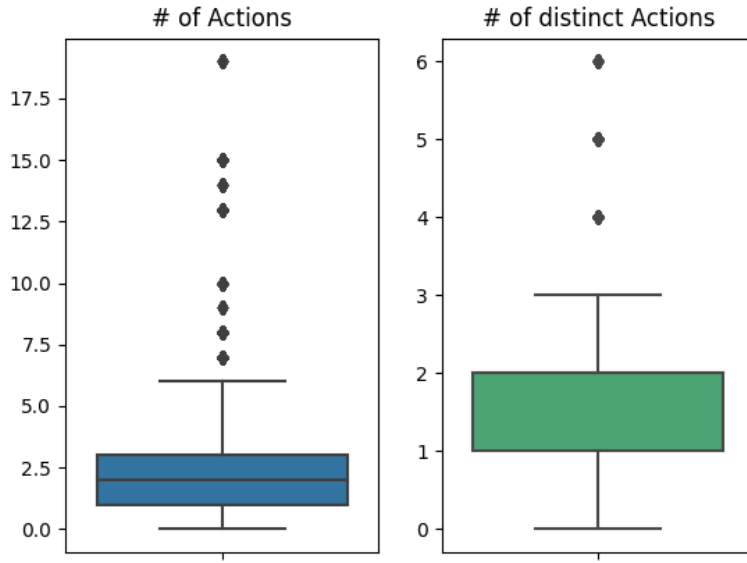


Figure 4.9: Distributions of the number of Actions per workflow file.

	# of Actions	# of distinct Actions
Minimum	0	0
Q1	1	1
Median	2	2
Q3	3	2
IQR	2	1
Maximum	19	6

Table 4.10: Number of Actions per workflow file.

From the above observations, it is observed that workflow files are generally using less than three (resp. two) Actions (resp. distinct Actions) available on the Marketplace. A manual observation of “outlier” workflow files using more than 15 Actions indicated that they were all using the same workflow file. The name

of the workflow is “CI” and this workflow is used to setup *Ruby*¹². Note that if a workflow file uses, for example, three Actions, it is possible that it is using the same Action three times. For this reason, the number of distinct Actions also has been counted, and reported on Table 4.10. Overall, 78 distinct Actions have been identified, and it has been observed that some Actions were used more often than others. Based on the 50,000 workflow files gathered for this question, it has been observed that the Action called *checkout* was the most used one, with 53,538 usages. This implies that Actions are sometimes used more than ones in a workflow file. The second Action is *setup-node* with 17,160 usages, and *setup-ruby* with 11,851 usages. The ten most popular Actions, along with their Marketplace categories, are listed in Table 4.11.

¹²<https://www.ruby-lang.org/en/>

Owner/Action	# of usages	Marketplace categories
actions/checkout	53,538	Utilities
actions/setup-node	17,160	Utilities
ruby/setup-ruby	11,851	Continuous integration
		Utilities
actions/upload-artifact	6,392	Utilities
actions/cache	3,892	Dependency management
		Utilities
actions/stale	3,620	Utilities
actions/setup-python	3,311	Utilities
codecov/codecov-action	2,735	Utilities
actions/setup-dotnet	2,310	Utilities
softprops/action-gh-release	1,793	Continuous integration
		Publishing

Table 4.11: Ten most used Actions in workflow files.

Table 4.11 reports the number of times Actions have been used in the observed workflow files. From this table it is observed that seven out of the ten most used Actions belong to the “utilities” Marketplace category. This table reports the number of times an Action appears in an observed workflow file. This might indicate a need for developers to use Actions from this category. While comparing the top ten listed in Table 4.11 with the top ten observed by Kinsman et al. [3] some similarities have been noted. The observations made for this question showed that six out of ten Actions are in common between the two observations (in a slightly different order), even if the repository selection has been made in different ways. This particularity can be observed in Table 4.12. The difference in the Actions used might be a consequence of the fact that the dataset used for

this question is different from the dataset using by Kinsman et al. [3], or it might indicate an evolution in the programming languages used by developers, or in their coding habits compared to three years ago. Moreover, eight out of ten Actions observed by Kinsman et al. [3] were from the “utilities” category. This might indicate that developers were already interested in Actions from this category back in 2019. Also, three Actions in the top ten of Kinsman et al. [3] are not part of the current top ten. In the case of *actions/setup-java* (resp. *actions/download-artifact*), this Actions reaches the 12th (resp. 24th) position of the current ranking. The *shivammathur/setup-php* is not part of the 78 distinct Actions observed in the 50,000 workflow files analyzed. Finally, the Actions “actions/setup-node”, “ruby/setup-ruby”, “actions/stale”, and “softprops/action-gh-release” were not present in the ranking provided by Kinsman et al. [3]. The Actions “actions/setup-ruby” is not maintained anymore, and users are requested to use “ruby/setup-ruby” which is maintained by the official Ruby organization. For the three others (“actions/setup-node”, “actions/stale”, and “softprops/action-gh-release”) they were all released on August 2019, implying that their absence from the top ten observed by Kinsman et al. [3] might be a consequence of their unpopularity at that time.

Current top ten	Kinsman et al. top ten
actions/checkout	actions/checkout
actions/setup-node	actions/setup-python
ruby/setup-ruby	actions/cache
actions/upload-artifact	actions/upload-artifact
actions/cache	actions/setup-java
actions/stale	actions/download-artifact
actions/setup-python	shivammathur/setup-php
codecov/codecov-action	actions/setup-ruby
actions/setup-dotnet	codecov/codecov-action
softprops/action-gh-release	actions/setup-dotnet

Table 4.12: Comparison between Kinsman et al. [3] top ten and the current top ten.

Another observation is the number of occurrences of each of the 78 distinct Actions used in the 50,000 workflow files. It has been observed that 50% of Actions occurred less than 99 times in all workflow files, and 75% of Actions were used less than 371 times in all workflow files. The statistics for the number of occurrences for each Actions are listed in Table 4.13, and it is represented as a box plot in Figure 4.10. To make this figure more readable, outliers values have been removed.

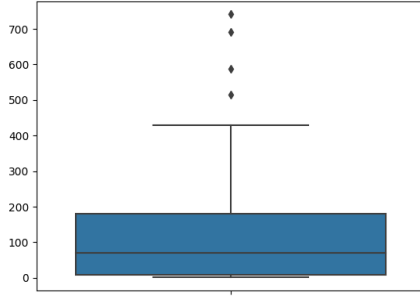


Figure 4.10: Distribution of the occurrences of Actions in the overall workflow files.

Q1	16
Median	95.5
Q3	310.25
Minimum	1
Maximum	53,538
IQR	294.25

Table 4.13: Actions occurrences in the overall workflow files.

50,000 workflow files have been gathered from random GitHub repositories. From these files, it has been observed that the majority of workflow files were using less than three Actions available on the Marketplace, which could indicate that developers are making workflow files for specific tasks, instead of having a single workflow file. Also, very few (2.4%) files were not using Actions available on the Marketplace. For the others, 78 distinct Actions have been identified. By observing the number of uses of each Action, it was observed that some actions (e.g. *checkout*) appeared multiple times in the same workflow file, and that some Actions were more used than others. The fact that an Actions appears multiple times on the same workflow file might indicate that the file is defining jobs for multiple triggers (e.g. push and pull requests). The most used Action was *checkout* with 53,538 usages, and the most represented category in the top then of most used Actions indicate that developers might be more interested by Actions from the “utilities” category. The top ten of most used Actions that has been observed for this question and the top ten observed by Kinsman et al. [3] shared similarities. Six out of ten Actions are common to both lists in a slightly different order. This might indicate that

these Actions did not decline in popularity since they made their observations. For the four others, the fact that they are not the same might indicate that developers have gain interest for other Actions, maybe because of the evolution of programming languages, or because of the evolution of their programming habits. Moreover, eight Actions in the top ten observed by Kinsman et al. [3] were in the “utilities” category, indicating that developers were already interested in Actions from this category.

Finally, it has been observed that the median for the number of uses per Actions was 95.5, and 75% of Actions were used less than 310.25 times, meaning that a few number of Actions were actually used a large number of times.

4.5 RQ5: What Are the Most Popular Actions?

This question aims to identify the most popular Actions on the Marketplace as well as the categories to which they belong. Knowing what are the most popular Actions will give us an idea about what are Actions the most needed for by developers. To answer this question it has been necessary to define what is a popular Action. While browsing the Marketplace, one can realize that GitHub determines the popularity of the Actions based on the number of stars that developers assigned to their GitHub repositories. According the the GitHub official documentation [23], stars are used by developers to make a repository easy to find later, since repositories that a developer starred are available by consulting the `https://github.com/stars` URL. Stars could be seen as bookmarks. The documentation states that the popularity of repositories is based on the number of stars they have. This way, the Action called “TruffleHog OSS”¹³ is considered as the most popular one on the Marketplace, with 8.8K stars. This Action can be used to find leaked credentials on a repository. Figure 4.11 shows what are

¹³<https://github.com/marketplace/actions/trufflehog-oss>

the eight most popular Actions available on the GitHub Marketplace. This way of ranking does not take in consideration other interactions that developers could make with the repositories of the Actions. Such interactions are:

- **Forking the repository of the Action.** According to the official GitHub documentation [23], a fork is a copy of a repository, allowing the developer who made the fork to make changes freely on the copy, without affecting the original repository. The documentation states that the two most common usages of forks are to propose changes to someone else’s repository to which the developer that made the fork does not have write access, or to use the forked project as the starting point for another idea.

Therefore, the number of forks on the repository of an Action shows some interest for this Action. Either because developers are willing to contribute to the Action, or because the Action could be used to create something else.

- **Watching the repository of the Action.** According to the official GitHub documentation [23], developers can watch another developer’s repository in order to get notified of activities in the repository. The documentation indicates that developers can customize the notifications they are getting, including notifications when participating or being mentioned, issues, pull requests, releases, discussions, and security alerts.

Because of this, the number of watchers on the repository of an Action might indicate some interest in this Action. The fact that developers are willing to get notified about changes, might indicate that this Action is somewhat interesting to them.

- **Contributing to the Action.** Contributing to an Action might be the clearest way of showing interest to this Action. Therefore, the number of

contributors on the repository of an Action might give an indication about the popularity of this Action.

- **Using the Action.** A developer can use an Action on its own GitHub repository workflows. One can accept that the more an Action is being used, the more this Action is popular.

There is no universal way to compute a popularity score, mainly because such a score depends on the available metrics, and on what is defined by “popular”. Even if GitHub chose to determine the popularity of Actions by their number of stars, this research question uses another approach to determine the popularity of Actions. This approach makes usage of the five available metrics that can be found on every GitHub repository, and which can attest to the popularity of the Actions. Such metrics are the number of stars, forks, watchers, contributors, and usages (known as dependents). It was chosen to sum the different metrics to determine the most popular Actions. Each information has been multiplied by a number ranging from zero to one in order to give more weight to the information that was considered to require more involvement from the developers. Table 4.14 gives the weights associated with each considered events on a GitHub repository.

Events	Weights
Dependents	0.05
Stars	0.1
Watchers	0.2
Forks	0.25
Contributors	0.4

Table 4.14: Weights for each events


Sort: Most installed/starred

Actions


An entirely new way to automate your development workflow.

14146 results for "sort:popularity-desc" filtered by Actions


Actions




TruffleHog OSS
By truffelsecurity
Scan Github Actions with TruffleHog
☆ 8.8k stars




Super-Linter
By github
It is a simple combination of various linters, written in bash, to help validate your source code
☆ 8k stars




Metrics embed
By lowlighter
An infographics generator with 40+ plugins and 200+ options to display stats about your GitHub account
☆ 7.6k stars




yq - portable yaml processor
By mikefarah
create, read, update, delete, merge, validate and do more with yaml
☆ 6k stars




Deploy to GitHub Pages
By Jameslves
This action will handle the deployment process of your project to GitHub Pages
☆ 3k stars



Cache
By actions
Cache artifacts like dependencies and build outputs to improve workflow execution time
☆ 3k stars



fastpages: An easy to use blogging platform with support for Jupyter Notebooks.
By fastai
Converts Jupyter notebooks and Word docs into Jekyll blog posts
☆ 2.7k stars



Checkout
By actions
Checkout a Git repository at a particular version
☆ 2.6k stars

Figure 4.11: Actions listed by popularity on the Marketplace.

Since being a contributor has been considered to require more involvement than making a forks of the repository of an Action, watching the repository of an Action, starring the repository of an Action, or using an Action, it is the one that has been allocated the greater weights. This reflection goes for the other metrics,

and the involvement is ranked as contributing > forking > watching > staring > using.

As previously stated, the popularity score for an Action is computed as the sum of the different metrics:

number of 0.05 * dependents + 0.1 * number of stars + 0.2 * number of watchers + 0.25 * number of forks + 0.4 * number of contributors.

It is to be noted that this formula has been established for this master's thesis and does not constitute a general solution to compute popularity based on well known values. The weights have been chosen in regard to the involvement the associated metric represent. No major impact was observed on the list of the most popular Actions by changing these values, keeping in mind that the weight for depends is always smaller than the other weights, etc. If the reader is interested in computing a popularity score knowing the number of positive rating and overall number of ratings, refer to the online article written by Evan Miller [24], and the article written by Edwin B. Wilson [25]. The solution presented by these articles could not be used for this master's thesis since GitHub does not provide a system for positive/negative ratings for repositories.

Computing the popularity score for every Action in the set of Actions collected for this work, the ten most popular Actions have been identified and listed in Table 4.15, along with their owner and Marketplace categories. It has been observed that 177 out of the 1,000 most popular Actions available on the Marketplace originate from the *continuous integration* Marketplace category. This category is followed by the *utilities* category with 169 out of 1,000 Actions originating from it, and *code-quality* with 153 Actions. The five most popular categories are listed in Table 4.16, and one can notice that these five categories are all in the top seven categories we can observed on Figure 4.5. This might indicate that popular categories have more Actions than the other categories. However, the most

populate category, i.e. “publishing” is not part of the top five of the most popular Actions. Actually, “publishing” is ranked at the 6th position with 136 Actions. Therefore, it is reasonable to say that there is a link between the number of Actions in a category, and the popularity of this category.

Also, the owners “Azure”, “reviewdog”, and “actions” were being identified as the three most popular ones, with 24, 19, and 15 popular Actions respectively developed by these users out of the 1,000 most popular Actions available on the Marketplace. The reason why Actions developed by these owners are popular might be because “Azure” is the official Microsoft account for Azure, “actions” are the official Actions developed by GitHub, and “reviewdog” is the account for Actions developed by reviewdog¹⁴. The five most popular owners are listed in Table 4.17. Along with the most popular categories, some categories were observed to be less popular, with less than 10 Actions per category. This is the case for *ides* with six Actions out of the 1,000 most popular ones, *learning* with five Actions, and *localization* with four Actions. This might indicate that developers do not have many needs for Actions in these categories. Moreover, 727 distinct owners have been identified among the 1,000 most popular Actions, with 614 (84.5%) of them owning one single Action.

¹⁴<https://github.com/reviewdog/reviewdog>

Action	Owner	Categories	Score
checkout	actions	utilities	178,917.55
setup-node-js-environment	actions	utilities	42,958.40
upload-a-build-artifact	actions	utilities	28,106.45
cache	actions	dependency management utilities	26,059.05
setup-python	actions	utilities	25,587.45
setup-java-jdk	actions	utilities	15,896.65
setup-go-environment	actions	utilities	10,591.00
download-a-build-artifact	actions	utilities	9,495.50
docker-login	docker	continuous integration utilities	8,097.15
build-and-push-docker-images	docker	continuous integration	7,953.25

Table 4.15: The ten most popular Actions.

Category	# of Actions
continuous integration	177
utilities	169
code quality	153
deployment	149
code review	143

Table 4.16: Most popular categories.

Owner	# of Actions
Azure	28
reviewdog	24
actions	15
moneyforward	12
microsoft	10

Table 4.17: Most popular owners.

Observing Table 4.15, one can directly see that one Action has a much higher popularity score than the others. This is explained by the number of usage of the *checkout* Action (more than 3,000,000). This Action is an official GitHub Action, meaning it has been developed by GitHub.

A formula based on the number of stars, forks, dependents, contributors, and watchers on the repository of each Actions in the collected set of Actions available on the Marketplace has been defined in order to identify the most popular Actions. Observing the 1,000 most popular Actions, it has been noted the *checkout* Action, which is an official GitHub Action, had a much higher popularity score than the others.

The most popular categories of Actions have been identified to be *continuous integration*, *code review*, and *utilities*. The most populated categories were also the most popular ones, even if in a slightly different order, confirming the hypothesis made in RQ2. Also, the most popular owners have been identified to be “Azure”, “reviewdog”, and “actions”. The most popular categories might indicate that developers are more interested in Actions falling in these categories. Also, the less popular categories represented among the 1,000 most popular Actions have been identified as *ides*, *learning*, and *localization*, indicating that developers have less needs for Actions falling in these categories.

4.6 RQ6: Are There Many Actions With Known Unresolved Issues?

This question aims to get an idea on how well Actions are maintained. One could think that a well maintained Action is less likely to have security flaws, or to have bugs. Therefore, knowing the proportion of well maintained Actions will give us an idea on the number of Actions that can potentially cause problems. This observations for this research question are based on the set of Actions gathered on July 31, 2022, and that were available on the Marketplace.

The notion of issue is specific to GitHub repositories. Referring to the official

GitHub documentation about issues [26], issues can be used to track ideas, give feedback, or give an explanation about a bug encountered. An issue is composed of a title and a description section. When an issue is opened, other developers can comment about the issue, and possibly a developer can give it a solution. This way, the owner of the repository can be informed by other developers about improvements they would like to see or bugs they are encountering. When the owner of the repository consider the issue to be addressed, either because the improvement has been made (or will not be, this choice is at the discretion of the owner of the repository), or because the bug has been fixed, they can close the issue. The lists of open and closed issues are available so that developers who encounter the same problem as a previously encountered one can access their resolution. Figure 4.12 illustrates an issue opened by a developer (*rtsisyk*). This developer gave an explanation about his issue and other developers (*albaars* and *gh2o*) gave him insights on the way to fix it. Note that the description of the issue has been cropped to fit on the page.

The time between the creation and the resolution of issues has been used to determine if an Action is well maintained or not. If an Actions has a majority of issues being closed within two months, the Actions is considered as well maintained. Otherwise, it is considered poorly maintained. Indeed, if issues are opened and never closed, or if issues are taking a long time to be closed, it might indicate a lack of maintenance from the owner of the Action. A period of two months has been chosen in order to give time to repository owners to fix the issue.

A first observation showed that 2,968 out of 8,114 (36.58%) Actions do not have any issues. This might indicate that these Actions are not popular, not used much, or flawless which is unlikely. These Actions are therefore excluded from the following observation, because it is not possible to deduce their level of maintenance based on their issues.

Broken submodule kills self-hosted runners completely #590



rtsisyk opened this issue on Sep 22, 2021 · 2 comments



rtsisyk commented on Sep 22, 2021

Tip ...

1. Add a submodule which points to missing commit
2. Run GH actions using a self-hosted runner
3. The self-hosted runner will be completely handicapped after that - all further tasks will fail, even from different branches:

```
[REDACTED]
Run actions/checkout@v2
with:
  repository: xxx/yyy
  token: ***
  ssh-strict: true
```

[Cropped]

```
Setting up auth
/usr/bin/git config --local --name-only --get-regexp core.sshCommand
/usr/bin/git submodule foreach --recursive git config --local --name-only --get-regexp 'core.sshCommand' && git config
Error: fatal: No url found for submodule path 'broken/submodule' in .gitmodules
Error: The process '/usr/bin/git' failed with exit code 128
```

To clarify - you need to run CI on this branch ONCE and all further runs on ALL branches will fail.

11



anonymouse64 mentioned this issue on Sep 22, 2021

tests: Include the tools from snapd-testing-tools project in "STESTTOOLS"
snapcore/snapd#10825

Merged



albaars commented on Nov 16, 2021

Tip ...

We ran into the same issue, except that instead of adding a submodule that references a non-existing commit we accidentally committed a submodule reference in a pull request without having the submodule mentioned in `.gitmodules`.

The problem is that the `actions/checkout` action stays on the broken commit forever because and gets stuck in the "setup auth" step which will fail for every subsequent job. I think this problem can be avoided by splitting the setup auth steps for the main repo from the setup auth steps of the submodules. Something like:

- main repo: setup auth
- main repo: fetch and checkout ref
- submodules: setup auth
- submodules: update

I think this way the action can switch away from the broken commit and recover.

1



gh2o commented on Nov 16, 2021

Tip ...

I've been working around the issue by running this command before the checkout step:

```
git checkout -f $(git -c user.name=x -c user.email=x@x commit-tree $(git hash-object -t tree /dev/null) < /dev/null) || :
```

This sets HEAD to a dummy commit before checking out the tree. The checkout step will then do a `git clean` which will wipe the tree before the auth step, preventing this issue from occurring again.

7

Figure 4.12: Example of issue with comments.

An Action will be considered as well maintained when the number of issues closed within two months is greater than the sum of issues closed in more than

two months and open issues.

From the remaining 5,146 Actions in the set used to answer this question, it was observed that 1,980 (38.48%) were considered as well maintained. This might mean that the minority of the owners of Actions have a tendency to process requests made through the issue system within two months. On the other hand, 3,166 Actions were considered as not well maintained. This might be because some Action doing the same thing in a better way has been released, or this might be because developers changed their habits and do not need these Actions anymore in their workflows.

The time between the moment an issue is open and closed has been computed to get an idea of how long it takes to owners to close an issue. The median was computed and is 290,404.5 seconds which represents three day and eight hours. Further more, 75% of issues were resolved in less than 2,764,913.5 seconds, which represent 32 days. The distribution of seconds between the moment an issue is open and closed is represented by Figure 4.13, and the values characterizing this distribution are listed in Table 4.18. This indicate that the majority of issues are closed quickly, reflecting a will of the developers to close them.

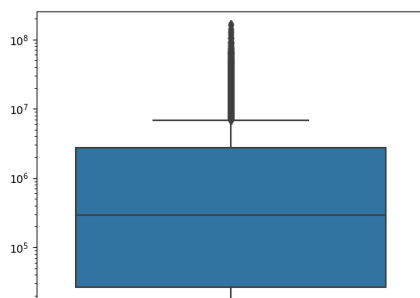


Figure 4.13: Distribution of seconds between the moment an issue is open and closed.

Minimum	0
Q1	26,194.75
Median	290,404.5
Q3	2,764,913.5
IQR	2,738,718.75
Maximum	165,679,697

Table 4.18: Distribution of seconds between the moment an issue is open and closed.

Furthermore, all issues were not closed. For this reason, the time elapsed between the moment they were opened and the current date (August 17, 2022) has been computed. The median of seconds since an issue is open (but not closed) is 38,860,652, which represents 449 days, and 75% of issues where open less than 56,732,252.25 seconds (656 days). This might indicate that some issues are “forgotten” by developers, or that some Actions are not maintained anymore. The distribution of seconds between the moment an issue is open and the current time is pictured by Figure 4.14, and its characterizing values are listed in Table 4.19.

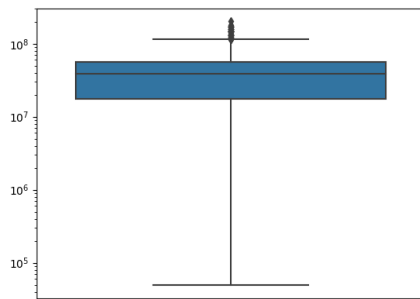


Figure 4.14: Distribution of seconds between the moment an issue is open and the current date.

Minimum	48,916
Q1	17,403,052
Median	38,860,652
Q3	56,732,252.25
IQR	39,329,200.25
Maximum	201,895,675

Table 4.19: Distribution of seconds between the moment an issue is open and the current date.

Looking at the 8,114 Actions gathered on July 31, 2022, it has been observed that only 1,030 (12.69%) Actions had closed issues but no open ones. This might indicate some sort of difficulty for developers to close all open issues for their Actions. However, only 787 Actions had open issues but no closed ones, which might indicate that developers are willing to close issues for their Actions.

The time between the creation and the resolution of issues has been computed to determine if an Action is well maintained or not. Actions without issues have been excluded from this research question, since it is not possible to deduce their level of maintenance base on their issues.

Actions have been considered as well maintained when the number of issues closed within two months is greater than the number of other issues, i.e. not closed issues and issues closed in more than two months.

A minority of Actions were considered as well maintained, indicating that the minority of owners are preoccupied by closing open issues. On the other hand, the majority of Actions were not well maintained, implying that these Actions might be doing the same thing as another Action but in a less efficient way, or this might imply that developers changed their habits and do not need these Actions anymore.

Finally, the time between the creation and the resolution of issues has been computed. The median time to close an issue is 290,404.5 seconds (three days) and the median time since issues were opened and not closed is 38,860,652 (449 days). This might imply that, when an issue is open, either it is quickly closed, or it will take a long time.

4.7 RQ7: Which Proportion of Actions Are Developed by Verified Users?

This question aims to observe the proportion of verified users and Actions on the Marketplace. Verified users are distinguished from other users by the presence of a badge located next to their username on the Marketplace, as pictured by Figure 4.15, or by the presence of the same badge on the Marketplace description page of Actions owned by verified users, as illustrated by Figure 4.16. The official GitHub documentation about badges on the Marketplace [27] indicates that creators with such a badge near their username had their identity verified by GitHub. Another page from the official GitHub documentation [28] states that using an Action in a workflow by specifying its tag must be done only if the user trusts the owner

of the Action. There is a risk because a tag can be moved or deleted, that can lead to the introduction of unwanted code in the workflow of developers using the Action. For this instance, the “verified badge” gives the user an indication of whether to trust the owner of the Action. Therefore, the fact that a user is verified might be considered as a sign of trust by developers willing to use some Actions from the Marketplace. Knowing the proportion of Actions developed by verified users might give an indication on the percentage of trustful Actions present on the Marketplace.

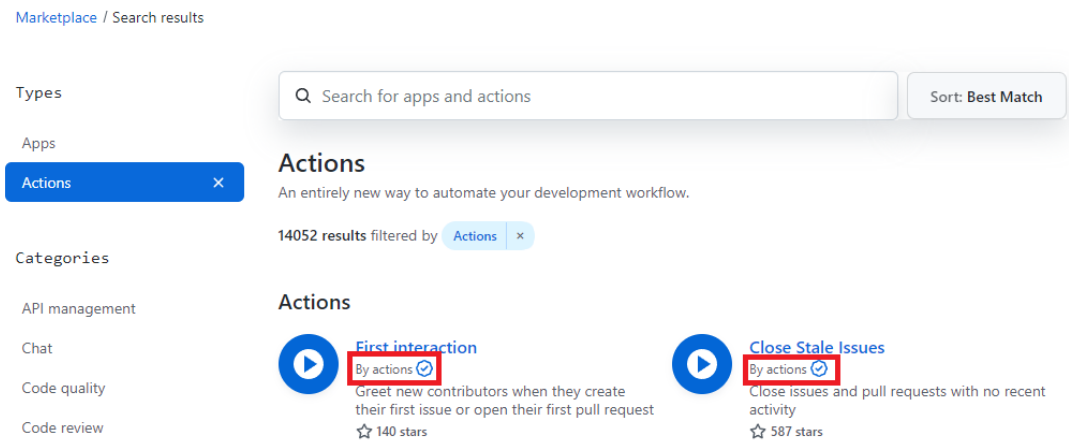
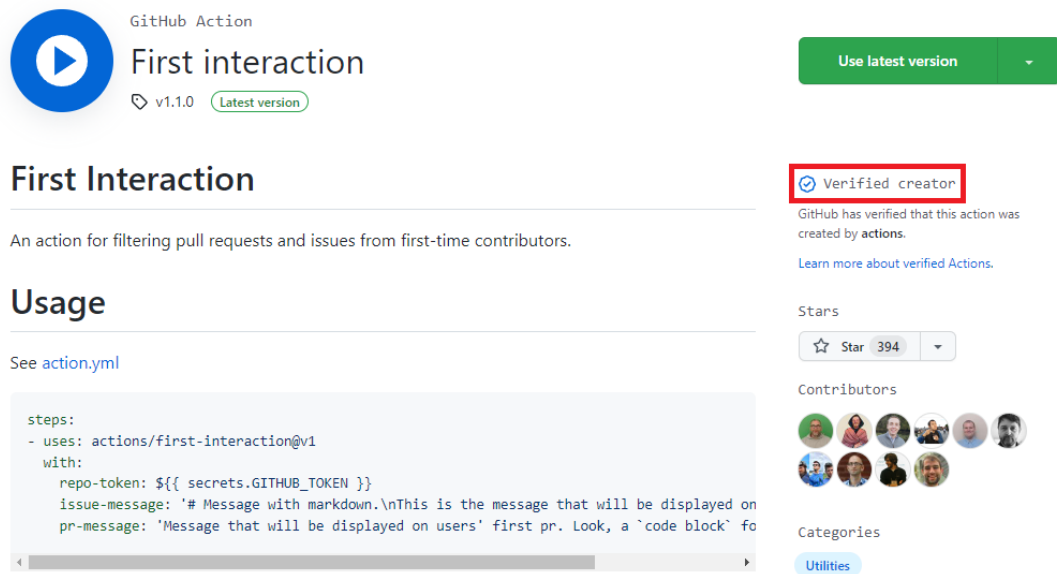


Figure 4.15: Verified badge on the Marketplace.



GitHub Action

First interaction

v1.1.0 Latest version

[Use latest version](#)

First Interaction

An action for filtering pull requests and issues from first-time contributors.

Usage

See [action.yml](#)

```
steps:
- uses: actions/first-interaction@v1
  with:
    repo-token: ${{ secrets.GITHUB_TOKEN }}
    issue-message: '# Message with markdown.\nThis is the message that will be displayed on'
    pr-message: 'Message that will be displayed on users' first pr. Look, a `code block` fo'
```

Verified creator

GitHub has verified that this action was created by actions.

[Learn more about verified Actions.](#)

Stars

☆ Star 394

Contributors

Categories

Utilities

Figure 4.16: Verified badge on one Action page.

Analyzing the set of 8,114 Actions gathered on July 31, 2022, it appeared that 7,772 Actions out of 8,114 were developed by unverified users. It represents 95.78% of the overall Actions on the Marketplace. Moreover, 5,925 distinct owners of Actions were identified on this set. Among these owners, 5,819 were identified as being unverified, which represents 98.2% of the owners. Those values are listed in Table 4.20. These observations might indicate that users are not looking to get verified by GitHub, and users might not consider verification as a strong trust factor.

	Number of unverified	Overall number	Percentage of unverified
Actions	7,772	8,114	95.78%
Owners	5,819	5,925	98,2%

Table 4.20: Number of unverified, overall number, and percentage of unverified Actions/users on the Marketplace.

Another goal is to understand how verification impacts the popularity¹⁵ of an Action. The 100 most popular Actions were observed to identify the proportion of Actions owned by verified users. It has been observed that 42 out of the 100 most popular Actions were developed by verified users. Hence, the majority of popular Actions are owned by unverified users, reinforcing the idea that users might not consider verification as a strong trust factor.

Analyzing the set of 8,114 Actions gathered on July 31, 2022, the proportion of Actions owned by verified users for each category on the Marketplace has been observed. It appeared that the highest percentage of Actions owned by a verified user for a category was 15.66% (category *continuous-integration*), which is also the most popular category observed in RQ5. However, this category is directly followed by the *security* category with 13.45% of Actions owned by verified users, which is not one of the most popular categories observed in RQ5, and the *deployment* category with 10.35% of Actions owned by verified users. The five categories with the highest proportion of Actions owned by verified users are listed in Table 4.21. The fact that these categories have a more important percentage of Actions owned by verified users might indicate a need for users to trust Actions from these categories.

Category	# of verified	Total	Percentage
Continuous integration	156	996	15.66%
Security	90	669	13.45%
Deployment	103	995	10.35%
Ides	3	33	9.09%
Utilities	67	998	6.71%

Table 4.21: Number of verified Actions per category.

¹⁵Popularity as defined in RQ5.

The large majority of Actions on the Marketplace are owned by unverified users. Moreover, unverified users represent more than 98% of the overall users on the Marketplace. Another observation showed that 42% of the 100 most popular Actions are owned by verified users. This high proportion of unverified users, hence Actions developed by unverified users, might indicate that developers are not willing to get verified, or users are not seeing verification as a strong trust factor, but might consider the popularity of an Action as a stronger metric.

Finally, it has been observed that the most popular category observed in RQ5 was the category with the highest proportion of Actions owned by verified users. Moreover, the categories “*continuous integration*”, “*security*”, and “*deployment*” are the categories with the highest proportion of verified Actions. A higher proportion of Actions owned by a verified user in a category might indicate a need of trustful Actions by users in this category.

4.8 RQ8: How Are Actions Triggered?

This question aims to get an idea on how developers are using Actions on their workflows, and check whether there is a relation between the trigger and the category of Actions. Knowing how Actions are triggered will give us a better understanding about how developers are using Actions in their workflow files. This question makes observations based on the set of 50,000 workflow files gathered for RQ4.

It has been observed that Actions available on the Marketplace were set to be triggered 274,257 times, meaning that an overall of 274,257 non distinct Actions were used in the set of 50,000 workflow files. The trigger “push” was used 128,281 times, directly following by “pull-request” used 110,666 times. Referring to the

GitHub Guides [29], a *commit* is described as a snapshot of the GitHub repository on which they are made, at a specific time. Referring to the same guide, a *push* is described as the action of updating the remote repository with local commits. Finally, a *pull request* is described by the GitHub documentation [23] as a way to inform others about changes pushed in a repository on GitHub. When a pull request is opened, developers can discuss and review the changes that have been pushed and add follow-up commits before the changes are merged into the repository.

The *pull request* and *push* triggers represent 87.12% of all triggers found in the workflow files. This might indicate that developers are more interested in automating pushes and pull-requests in their workflows. Overall 13 distinct triggers have been identified. The list of the five most used triggers along with the number of times they have been used is presented by Table 4.22. The trigger “pull” was only used two times, indicating a very low interest for developer to automate actions when doing a pull on their repositories, which was expected because no interaction is expected when a developer makes a “pull” in a repository. This trigger is described by GitHub Guides [29] as the way to locally synchronize a GitHub repository.

Trigger	# of uses
push	128,281
pull_request	110,666
workflow_dispatch	23,058
schedule	7,429
release	1,324

Table 4.22: Most used triggers in workflow files.

Another observation has been made in order to identify a potential relationship

between the types of triggers and the category of Actions in which they are the most used. It has been observed that the five most used triggers were used with Actions falling in the “utilities” category. This is not surprising since Actions in this category are used to automate steps of the developers’ development workflow. The second category of Actions that used the triggers “push” and “pull_request” the most was “continuous integration”, which contains CI-related Actions. This observation indicated that developers are more interested in automating their interactions with their repositories using GitHub Actions when it comes to doing “push” and “pull_request”.

Among the workflow files observed for this RQ, Actions were triggered 274,257 times, and 13 distinct triggers were identified with “push” and “pull-requests” being the most used ones (87.12% of all triggers). This observation might indicate that developers are more interested in automating these interactions with their repositories.

The five most used triggers (*push*, *pull request*, *workflow dispatch*, *schedule*, and *release*) were all used with Actions falling in the “utilities” category. This indicated that developers were more interesting in automating their interactions with their repositories when “pushes” and “pull requests” occurred.

Chapter 5

Conclusion

A series of observations have been made about GitHub Actions. The evolution of the GitHub Marketplace over time has been analyzed and the number of Actions available on the Marketplace tended to grow. By looking at the proportion of Actions in each categories, it appeared that Actions some categories had more Actions than other, e.g. *publishing* and *code quality*.

Based on the Actions releases names and dates, we observed that patches were the most common release type, and the one that take the least time to switch from one version to a new one. Next, by consulting the workflows files of GitHub repositories, we identified that the majority of workflow files were using less than three Actions. Still by consulting the workflow files, we determined that developers were more interested by automating the workflow when it comes to push and pull requests.

Then, we determined which Actions were the most popular. Also, the most popular categories have been identified. The popularity of Actions and categories indicated an interest for developers to have these Actions.

After that, the time between the issues of Actions has been observed. Observation suggested that either an issue will be closed within two months, or it

will take a long time to close it. Also, a minority of Actions were considered well maintained, meaning that using the majority of Actions on the Marketplace might imply taking a risk for potential problems.

Finally, we determined the proportion of verified users on the Marketplace. It appeared that the majority of them were unverified, implying that users might not consider the verification as a strong trust factor, but might consider the popularity as a better trust factor.

Some problems were encountered during the development of the data extraction tool. The use of the REST API has been changed (when possible) in order to use the GraphQL API. This update decreased the number of requests to the API because it was necessary to send multiple requests to the REST API in order to get the same information it was possible to get issuing a single request to the GraphQL API. Also, it is unclear why, but GitHub only showed more or less half the Actions available on the marketplace. Future studies might try to find another way to find distinct Actions.

Throughout this work it appeared that there were many observations to make about GitHub Actions. Some questions that would follow the continuity of this work could be:

- Is there a lot of “code duplication” between different actions? This question has been discussed with my thesis director. Because we did not find a reliable tool to detect code duplication in YAML files, this question was left unanswered.
- What proportion of repositories are using GitHub Actions? This question has been studied in Kinsman et al. paper [3]. A new analysis would allow to observe the evolution since their study.
- Are all Actions on a repository used? To answer this question it is possible to analyze the workflow runs of a repository. Because of the time constraint imposed for this work, this question was left unanswered.

Bibliography

- [1] Mehdi Golzadeh, Alexandre Decan, and Tom Mens. On the rise and fall of ci services in github, Mars 2022.
- [2] Jiang Bo, Zhang Zhenyu, Chan W.K., Tse T.H., and Chen Tsong Yueh. How well does test case prioritization integrate with statistical fault localization?, February 2012.
- [3] Timothy Kinsman, Mairieli Wessel, Marco A. Gerosa, and Cristoph Treude. How do software developers use github actions to automate their workflows?, Mars 2021.
- [4] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining github, May 2014.
- [5] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: Transparency and collaboration in an open software repository, February 2012.
- [6] Thung Ferdian, F. Bissy Tegawende, Lo David, and Jiang Lingxiao. Network structure of social coding in github, 2013.

- [7] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github, August 2015.
- [8] Dustin Smith, Daniella Vallalba, Michelle Irvive, Dave Stanke, and Nathen Harvey. Accelerate state of devops 2021, 2021.
- [9] GitHub. Ci/cd explained. <https://resources.github.com/ci-cd/>, 2021. Consulted on July 4, 2022.
- [10] GitHub. Understanding github actions. <https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions>, 2022. Consulted on May 27, 2022.
- [11] GitHub. Reusing workflows. <https://docs.github.com/en/actions/using-workflows/reusing-workflows>, 2022. Consulted on July 4, 2022.
- [12] CircleCI. Welcome to circleci documentation. <https://circleci.com/docs>, 2022. Consulted on July 5, 2022.
- [13] Travis-CI. Travis ci. <https://www.travis-ci.com/about-us/>, N/A. Consulted on May 29, 2022.
- [14] GitHub. Publishing actions in github marketplace. <https://docs.github.com/en/enterprise-cloud@latest/actions/creating-actions/publishing-actions-in-github-marketplace>, 2022. Consulted on July 17, 2022.
- [15] Selenium. About selenium. <https://www.selenium.dev/about/>, 2022. Consulted on Avril 28, 2022.
- [16] Selenium. Install browser drivers. https://www.selenium.dev/documentation/webdriver/getting_started/install_drivers/, 2022. Consulted on July 5, 2022.

- [17] GitHub. About github's apis. <https://docs.github.com/en/developers/overview/about-githubs-apis>, 2022. Consulted on July 5, 2022.
- [18] Python. sqlite3 — db-api 2.0 interface for sqlite databases. <https://docs.python.org/3/library/sqlite3.html>, April 2022. Consulted on May 9, 2022.
- [19] Matplotlib. Matplotlib: Visualization with python. <https://matplotlib.org/>, 2021. Consulted on May 10, 2022.
- [20] Michael Waskom. seaborn: statistical data visualization. <https://seaborn.pydata.org/>, 2021. Consulted on May 10, 2022.
- [21] Corder Gregory W. and Foreman Dali I. *Nonparametric Statistics: A Step-by-Step Approach*. Wiley, 2014.
- [22] Bruce Peter C. *Introductory Statistics and Analytics*. Wiley, 2014.
- [23] GitHub. Github docs. <https://docs.github.com/en>, 2022. Consulted on July 18, 2022.
- [24] Evan Miller. How not to sort by average rating. <https://www.evanmiller.org/how-not-to-sort-by-average-rating.html>, february 2009. Consulted on July 18, 2022.
- [25] Wison Edwin B. Probable inference, the law of succession, and statistical inference, 1927.
- [26] GitHub. About issues. <https://docs.github.com/en/issues/tracking-your-work-with-issues/about-issues>, 2022. Consulted on July 3, 2022.

- [27] GitHub. About marketplace badges. <https://docs.github.com/en/developers/github-marketplace/github-marketplace-overview/about-marketplace-badges#for-github-actions>, 2022. Consulted on July 11, 2022.
- [28] GitHub. Security hardening for github actions. <https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions#using-third-party-actions>, 2022. Consulted on July 11, 2022.
- [29] GitHub. Git guides. <https://github.com/git-guides/>, 2022. Consulted on August 13, 2022.
- [30] Linda Erlenhov, Francisco Gomes De Oliveira, and Philip Leitner. An empirical study of bots in software development: Characteristics and challenges from a practitioner’s perspective, October 2020.
- [31] Mahmoud Alfadel, Diego Elias Costa, Emad Shihad, and Mouafak Mkhallati. On the use of dependabot security pull requests, February 2021.
- [32] Cochran William G. *Sampling Techniques*. Asian Publishing House, 1953.