Chapter 8
The GitHub Development Workflow Automation Ecosystems

Mairieli Wessel, Tom Mens, Alexandre Decan, and Pooya Rostami Mazrae

Abstract Large-scale software development has become a highly collaborative and geographically distributed endeavor, especially in open-source software development ecosystems and their associated developer communities. It has given rise to modern development processes (e.g., pull-based development) that involve a wide range of activities such as issue and bug handling, code reviewing, coding, testing, and deployment. These often very effort-intensive activities are supported by a wide variety of tools such as version control systems, bug and issue trackers, code reviewing systems, code quality analysis tools, test automation, dependency management, and vulnerability detection tools. To reduce the complexity of the collaborative development process, many of the repetitive human activities that are part of the development workflow are being automated by CI/CD tools that help to increase the productivity and quality of software projects. Social coding platforms aim to integrate all this tooling and workflow automation in a single encompassing environment. These social coding platforms gave rise to the emergence of development bots, facilitating the integration with external CI/CD tools and enabling the automation of many other development-related tasks. GitHub, the most popular social coding platform, has introduced GitHub Actions to automate workflows in its hosted software development repositories since November 2019. This chapter explores the ecosystems of development bots and GitHub Actions and their interconnection. It provides an extensive survey of the state of the art in this domain, discusses the opportunities and threats that these ecosystems entail, and

M. Wessel (✉)
Radboud University, Nijmegen, Netherlands
e-mail: mairieli.wessel@ru.nl

T. Mens · P. Rostami Mazrae
University of Mons, Mons, Belgium
e-mail: Tom.Mens@umons.ac.be; PooyaRostami.Mazrae@umons.ac.be

A. Decan
Alexandre Decan is a Research Associate of the Fonds de la Recherche Scientifique - FNRS,
University of Mons, Mons, Belgium
e-mail: Alexandre.Decan@umons.ac.be

reports on the challenges and future perspectives for researchers as well as software practitioners.

8.1 Introduction

This introductory section presents the necessary context to set the scene. We start by introducing collaborative software development and social coding (Sect. 8.1.1). Next, we report on the emergence and dominance of GitHub as the most popular social coding platform (Sect. 8.1.2). We continue with a discussion of the practices of continuous integration, deployment, and delivery (Sect. 8.1.3). Finally, we explain the workflow automation solutions of development bots and GitHub Actions that have emerged as highly interconnected ecosystems to support these practices and that have become omnipresent in GitHub (Sect. 8.1.4).

We argue that these workflow automation solutions in GitHub constitute novel software ecosystems that are worthy of being studied in their own right. More specifically, Sect. 8.2 focuses on how development bots should be considered as an integral and important part of the fabric of GitHub's socio-technical ecosystem. Section 8.3 focuses on GitHub Actions and how this forms an automation workflow dependency network bearing many similarities with the ones that have been studied abundantly for packaging ecosystems of reusable software libraries. Section 8.4 wraps up with a discussion about how both types of automation solutions are interrelated and how they are drastically changing the larger GitHub ecosystem of which they are part.

8.1.1 Collaborative Software Development and Social Coding

The large majority of today's software is either open source or depends on it to a large extent. In response to a demand for higher-quality software products and faster time to market, open-source software (OSS) development has become a continuous, highly distributed, and collaborative endeavor [15]. In such a setting, development teams often collaborate on these projects without geographical boundaries [37]. It is no longer expected for software projects to have all their developers working in the same location during the same office hours. To achieve this new way of software development, specific collaboration mechanisms have been devised such as issue and bug tracking, pull-based development [34], code reviews, commenting mechanisms, and the use of social communication channels to interact with other project contributors. Collaboration extends distributed software development from a primarily technical activity to an increasingly social phenomenon [55]. Social activities play an essential role in collaborative development and become sometimes as critical as technical activities. They also come with their own challenges, for

example, because of cultural differences, language barriers, or social conflicts [10, 38].

A multitude and variety of development-related activities need to be carried out during collaborative software development: developing, debugging, testing, and reviewing code; quality and security analysis; packaging, releasing, and deploying software distributions; and so on. This makes it increasingly challenging for contributor communities to keep up with the rapid pace of producing and maintaining high-quality software releases. It requires the orchestrated use of a wide range of tools such as version control systems, software distribution managers, bug and issue trackers, and vulnerability and dependency analyzers.

These tools therefore tend to be integrated into so-called social coding platforms (e.g., GitLab, GitHub, Bitbucket) that have revolutionized collaborative software development practices in the last decade because they provide a high degree of social transparency to all aspects of the development process [16]. Social coding platforms aim to reconcile the technical and social aspects of software development in a single environment. It offers the project contributors a seamless interface and experience to contribute with their peers in an open and fully transparent workflow, where users can contribute bugs and feature requests through an issue tracking system, external contributors can propose code changes through a pull request mechanism, core software developers can push (i.e., commit) their own code changes directly and accept and integrate the changes proposed by external contributors, and code review mechanisms allow code changes to be reviewed by other developers before they can be accepted [34].

8.1.2 The GitHub Social Coding Platform

GitHub has revolutionized software development since it was the first platform to propose a *pull-based software development* process [3, 34]. The pull-based model allows to make a distinction between direct contributions from a typically small group of core developers with commit access to the main code repository and indirect contributions from external contributors that do not have direct commit access. This allows external contributors to propose code changes and code additions through so-called pull requests (PRs). To do so, these contributors have to *fork* the main repository, update their local copies with code changes, and submit PRs to request to pull these changes into the main code repository [35]. This indirect contribution method enables the project's maintainers to review the code submitted through each PR, test it, request changes to the submitter of the PR if needed, and finally integrate the PR into the codebase without getting involved in code development [35]. A pull-based development process also comes at a certain cost, since it raises the need for integrators—specialized project members responsible for managing others' contributions who act as guardians of the projects' quality [36].

The focus of this chapter will be on GitHub, since it is the largest and most popular social coding platform by far, especially for open-source projects, and as a

consequence, it has been the focus of a significant amount of empirical research. It is a web-based platform on the cloud, based on the git version control system, that hosts the development history of millions of collaborative software repositories and accommodates over 94 million users in 2022 [29].

GitHub continues to include more and more support for collaborative software development such as a web-based interface on top of the git version control system, an issue tracker, the ability to manage project collaborators, the ability to have a discussion forum for each git repository, an easy way to manage PRs or even to submit new PRs directly from within the GitHub interface, a mechanism to create project releases, the ability to create and host project websites, the ability to plan and track projects, support for analyzing outdated dependencies and security vulnerabilities, and metrics and visualizations that provide insights in how the project and its community are evolving over time. GitHub also comes with a REST and GraphQL API to query and retrieve data from GitHub or to integrate GitHub repositories with external tools. By late 2022, GitHub added a range of new features including (i) github.dev, a web-based code editor that runs entirely in the Internet browser to navigate, edit, and commit code changes directly from within the browser; (ii) GitHub CodeSpaces, a more complete development environment that is hosted in the cloud; (iii) GitHub Packages to create, publish, view, and install new packages directly from one's code repository; (iv) GitHub CoPilot, an AI-based tool that provides smart code auto-completion; and (v) GitHub Actions, a workflow automation tool fully integrated into GitHub.

8.1.3 Continuous Integration and Deployment

Continuous integration (CI), continuous deployment, and continuous delivery (CD) have become the cornerstone of collaborative software development practices. CI practices were introduced in the late 1990s in the context of agile development and extreme programming methodologies. According to the Agile Manifesto principles, "our highest priority is to satisfy the customer through early and continuous delivery of valuable software" [5]. In their seminal blog [28], Fowler and Foemmel presented CI as a way to increase the speed of software development while at the same time improving software quality and reducing the cost and risk of work integration among distributed teams. They outlined core CI practices to do so, including frequent code commits, automated tests that run several times a day, frequent and fully reproducible builds, immediately fixing broken builds, and so on. CD practices, on the other hand, aim at automating the delivery and deployment of software products, following any changes to their code [12]. Key elements of continuous deployment are the creation of feasible, small, and isolated software updates that are automatically deployed immediately after completion of the development and testing [49].

Many self-hosted CI/CD tools and cloud-based CI/CD services automate the integration of code changes from multiple contributors into a centralized repository

where automated builds, tests, quality checks, and deployments are run. Popular examples of such CI/CD solutions are Jenkins, Travis, CircleCI, and Azure DevOps. They have been the subject of much empirical research over the last decades. An excellent starting point is the systematic literature review by Soares et al. [50], covering 106 research publications reporting on the use of CI/CD. This review aimed at identifying and interpreting empirical evidence regarding how CI/CD impacts software development. It revealed that CI/CD has many benefits for software projects. Besides the aforementioned cost reduction and quality and productivity improvement, it also comes with a reduction of security risks, increased project transparency and predictability, greater confidence in the software product, easiness to locate and fix bugs, and improved team communication. CI can also be beneficial to pull-based development by improving and accelerating the integration process.

CI/CD services have also been built into social coding platforms. With GitLab CI/CD, GitLab has already featured CI/CD capabilities since November 2012. Bitbucket has supported Pipelines since May 2016. Based on popular demand, in response to this support for CI/CD in competing social coding platforms, GitHub officially began supporting CI/CD through GitHub Actions in August 2019, and the product was released publicly in November 2019. Before the release of GitHub Actions, Travis used to be the most popular CI/CD cloud service for GitHub repositories [6]. However, quantitative evidence has revealed that Travis is getting replaced by GitHub Actions at a rapid pace [32]. Additional qualitative evidence has revealed the reasons behind this replacement and the added value that GitHub Actions is bringing in comparison to Travis [43].

8.1.4 The Workflow Automation Ecosystems of GitHub

The previous sections have highlighted that global software development, especially for OSS projects, is a continuous, highly distributed, and collaborative endeavor. The diversity in skills and interests of the projects' contributors and the wide diversity of activities that need to be supported (e.g., coding, debugging, testing, documenting, packaging, deploying, quality analysis, security analysis, and dependency analysis) make it very challenging for project communities to keep up with the rapid pace of producing and maintaining high-quality software releases.

Solutions to automate part of the software development workflow, such as the aforementioned CI/CD tools and services, have been successfully used to reduce this maintenance burden (see Sect. 8.1.3). However, these tools do not support the entire range of project-related activities for which automation could come to the rescue. There are many repetitive and time-consuming social and technical activities for which, traditionally, CI/CD tools did not provide any support. Some examples of these are welcoming newcomers, keeping dependencies up to date, detecting and resolving security vulnerabilities, triaging issues, closing stale issues, finding and assigning code reviewers, encouraging contributors to remain active,

and software licensing. To help project contributors in carrying out these activities, CI/CD solutions have been complemented by novel workflow automation solutions:

Development Bots A well-known and very popular example of such workflow automation solutions is what we will refer to as *development bots*. Erlenhov et al. [27] consider development bots to be artificial software developers who are autonomous and adaptive and have technical as well as social competence. Such automated software development agents have become a widely accepted interface for interacting with human contributors and automating some of their tasks. A study by Wang et al. revealed that bots are frequently used in the most popular OSS projects on GitHub [56]. These bots tend to be specialized in specific activities, belonging to the following main categories: CI/CD assistance, issue and PR management, code review support, dependency and security management, community support, and documentation generation. Section 8.2 will discuss in detail how such bots are used on GitHub to automate part of the software development workflow and how these bots form an integral part of the socio-technical ecosystem of software contributors and software projects. More specifically, bots affect the social interaction within a software project, as they influence how human contributors communicate and collaborate and may even change the collaboration patterns, habits, and productivity of project contributors [42].

GitHub Actions Another popular mechanism to automate development activities in GitHub repositories is using *GitHub Actions*, a workflow automation service officially released in November 2019. Its deep integration into GitHub implies that GitHub Actions can be used not only for automating traditional CI/CD services such as executing test suites or deploying new releases, but also to facilitate other activities such as code reviews, communicating with developers, and monitoring and fixing dependencies and security vulnerabilities. GitHub Actions allows project maintainers to define automated workflows for such activities. These workflows can be triggered in a variety of ways such as commits, issues, pull requests, comments, schedules, and many more [11]. GitHub Actions also promotes the use and sharing of reusable components, called *Actions*, in workflows. These Actions are distributed in public GitHub repositories and on the GitHub Marketplace.[1] They allow developers to automate their workflows by easily integrating specific tasks (e.g., set up a specific programming language environment, publish a release on a package registry, run tests, and check code quality) without having to write the corresponding code. Only 18 months after its introduction, GitHub Actions has become the most dominant CI/CD service on GitHub [32]. Section 8.3 presents this ecosystem of reusable Actions in more detail. This ecosystem forms a technical dependency network that bears many similarities with traditional package dependency networks of reusable software libraries (such as npm for JavaScript, RubyGems for Ruby, NuGet for .NET, Packagist for PHP, CRAN for R, Maven for Java) that have been the subject of many past empirical studies (e.g., [21]).

---

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

The two aforementioned workflow automation solutions are increasingly used in OSS projects on GitHub, partly because of their tight integration into the social coding platform, thereby effectively transforming the software development automation landscape. It therefore seems fair to claim that they form new *development workflow automation ecosystems* that are worthy of being investigated in their own right. Research on these ecosystems is still in its infancy, given the relative novelty of the proposed automation solutions. Development bots and workflows that rely on GitHub Actions are already used in hundreds of thousands of GitHub repositories, and their usage continues to increase (the Marketplace of GitHub Actions has been growing exponentially since its introduction), justifying the need for further studies on the evolution of these ecosystems and their impact on collaborative software development practices.

8.2 Workflow Automation Through Development Bots

As explained in Sect. 8.1.4, development bots emerged on social coding platforms such as GitHub to enable the automation of various routines and time-consuming tasks previously assigned only to human developers. This section explores how bots are an integral part of GitHub's socio-technical collaborative development ecosystem. Considering the workflow automation provided by development bots, we focus on the various usage scenarios, advantages, shortcomings, challenges, and opportunities of using them.

8.2.1 What Are Development Bots?

Development bots that reside on social coding platforms such as GitHub are often seen as workflow automation providers due to their ability to react to certain stimuli, such as events triggered by human developers or other tools, and automate routine development-related tasks in response. To a certain extent, bots may act autonomously [27, 64]. In open-source repositories, bots can leverage the public availability of software assets, including source code, discussions, issues, and comments.

Besides automatically executing activities, development bots may also exhibit human-like traits. Erlenhov et al. [27] describe bots based on their social competence, which varies from very simple identity characteristics (e.g., a human-like name or profile picture) to more sophisticated ones such as artificial intelligence and the ability to adapt to distinct scenarios. In practice, bots that are active in GitHub repositories are automated agents that interact with the GitHub platform in essentially the same way as a typical human developer would be expected to: they possess a GitHub account, commit code, open or close issues or PRs, and comment on all of the above. Some bots have an official integration with GitHub and

are publicly available as Apps in the GitHub marketplace.[2] These *official* bots are properly tagged as such in the various activities they make in the GitHub platform.

Bots can also be used as an interface between human developers and other software services, such as external CI/CD tools or other third-party applications. Such bots provide additional value on top of the services they offer an interface for, by providing new forms of interaction with these services, or by combining multiple services.

One particularly interesting example is Dependabot, a dependency management tool responsible for creating PRs in GitHub repositories to propose to upgrade dependencies in order to resolve or reduce the risk of security vulnerabilities or bugs. Dependabot acts as an interface between the project maintainer, who is responsible for keeping the project dependencies up to date, and the package managers (such as npm for JavaScript) that expose the reusable packages that the project depends on. While originally it used to be a third-party service, Dependabot is now deeply integrated into the GitHub platform and has become one of the most popular dependency management bots, accounting for more than 7.7 million dependency updates in OSS projects [65]. A well-known alternative is renovatebot.[3]

Bots can even create, review, and decide whether to integrate the changes made in a PR into the repository by themselves in complete autonomy. Figure 8.1 provides an example of multiple bots interacting as part of a single PR. There is not a single human contributor involved in this interaction. The PR is triggered by a recommendation by Dependabot to update a dependency. The mergify bot reacts to this by verifying if the proposed change passes all checks, and accepts and merges the PR. Finally, nemobot reacts with a visual comment applauding the merged PR.

From a research viewpoint, the increasing use of bots raises the need for large-scale empirical studies on bot usage in social coding platforms such as GitHub. Such studies enable us to assess whether bots serve their intended purpose and whether their introduction has any positive or negative side effects on the socio-technical fabric of the project or ecosystem in which they are used. To enable such empirical studies, it is necessary to determine which projects rely on bots and which user accounts actually correspond to bots. Several bot detection heuristics have been proposed to automatically identify bot contributions [1, 23, 30]. BIMAN [23] relies on bot naming conventions, repetitiveness in commit messages, and features related to files changed in commits. BoDeGHa [30] relies on comment patterns in issue and PR comments in GitHub repositories, based on the assumption that bots tend to use different and fewer comment patterns than humans. BotHunter [1] additionally relies on features corresponding to profile information (e.g., account name) and account activity (e.g., median daily activity) to identify bot accounts

---

[2] https://github.com/marketplace?type=apps.

[3] https://github.com/renovatebot/renovate.

Fig. 8.1 Example of multiple bots interacting within the same PR

more accurately. BoDeGiC [31] allows to detect bots in git repositories based on commit messages and has been trained using the classification model of BoDeGHa.

An important challenge when identifying automated contributions by bots is the presence of so-called *mixed accounts*—accounts used by a human developer and a bot in parallel—exhibiting both human-like and bot-like behavior [30]. Not properly detecting such cases is likely to lead to false positives and false negatives during bot detection, which may affect the outcome of empirical analyses. Cassee et al. [9] have shown that existing classification models are not suitable to reliably detect mixed accounts.

8.2.2   The Role of Bots in GitHub's Socio-technical Ecosystem

An important characteristic of bots is that they form an integral part of GitHub's socio-technical ecosystem of collaborative software development. To consider them as such, we adopt an *ecosystemic* and *socio-technical* viewpoint, similar to Constantinou and Mens [14] who viewed a software ecosystem as a socio-technical network that is composed of a combination of technical components (e.g., software projects and their source code history) and social components (e.g., contributor and communities involved in the development and maintenance of the software).

An interesting novelty of bots is that while they are technical components themselves (since they are executable software artifacts), they should also be considered as being social components, since they play a crucial role in the social aspects of the ecosystem. The assistance provided by bots, as new voices in the development conversation [45], has the potential to smooth and improve the efficiency of developers' communication. Wessel et al. [60] have shown that the number of human comments decreases when using bots, which usually implies that the number of trivial discussions decreases. Indeed, bots are meant to relieve, augment, and support the collaborative software development activities that are carried out by the human contributors that jointly develop and maintain large software projects. Moreover, bots often interact with human collaborators (and with other bots) using the same interface as humans do.

Figure 8.2 illustrates an exemplary case of the role that bots play in this socio-technical ecosystem. A human contributor submits a PR to add tests to a particular project module. The first bot to react to the PR, `changeset-bot`, verifies whether the changeset file was updated, and the proposed change will be released into a specific version of the packages implemented in the repository. Then, `vercel` bot deploys the code to the third-party application Vercel and provides a URL for the developers to inspect a deployment preview in the PR. Next, the `codesandbox-ci` bot provides the URL of an isolated test environment to validate the changes made in the PR. Finally, the human project maintainer approves the changes, reacts with a comment, and merges the PR.

Like the many roles human software developers can fulfill, a variety of bots have become highly active actors in every phase of the development automation

Fig. 8.2 Example of an interaction between two humans and three bots within a single PR

workflow. Thanks to the continuous efforts of practitioners and researchers, a wide range of development bots are available for use by developers [42, 56]. Wang et al. [56] have shown that bot usage is common practice in OSS development. Through repository mining of 613 GitHub repositories, they found 201 different bots. Similar to prior research by Wessel et al. [61], the authors provided a classification of bots according to their main role in the repository. These categories include CI/CD assistance, issue and PR management, code review support, dependency and security management, community support, and documentation generation.

In addition to the aforementioned examples of bots, other sophisticated bots have been proposed in the literature. Wyrich and Bogner [64], for example, proposed a bot that automatically refactors the source code of a project. Their goal was to eliminate the need for developers to manually find and correct code smells, as this task can be very time-consuming and may require certain expertise. Therefore, the bot was designed to act autonomously, integrating into the natural workflow of the development team on GitHub. The bot makes code changes corresponding to proposed code refactorings and submits a PR with these changes. Project maintainers can review these changes and decide to integrate them into the codebase.

8.2.3 Advantages of Using Development Bots

Development bots generally execute tasks that would otherwise have to be performed manually by humans. Through interviews with industry practitioners, Erlenhov et al. [26] found that bots are used either because they improve productivity or enable activities that are not realistically feasible for humans [26]. Some software practitioners stress that bots are able to carry out certain tasks better than humans due to their availability, scalability, and capacity to process large amounts of data [26]. For example, bots can handle tasks continuously 24/7 without ever needing to take a break. Song and Chaparro [51] designed BEE, a bot that automatically analyzes incoming issues on GitHub repositories and immediately provides feedback on them. Due to BEE's prompt reaction, issue reporters can more quickly gain a general idea of what is missing without waiting for the project maintainers' feedback. Bots also scale, increase consistency, and mitigate human errors. In terms of productivity increase, bot usage is frequently motivated by the necessity of spending less time on routine, time-consuming, or tedious tasks. Automating such activities through bots allows developers to focus on their core code development and review tasks [26, 53]. Mirhosseini and Parnin [44] analyzed automated PRs created by greenkeeper, a bot to update dependencies, similar to the ones created by dependabot. Such a bot avoids manually monitoring for new releases in the packages. The results show that OSS repositories that use the bot upgraded the dependencies 1.6 times more regularly than repositories that did not use any other bots or tools.

Specifically in the context of code reviews, Wessel et al. [59] carried out a survey with 127 software project developers to investigate the advantages of adopting bots to support code review activities. Their study confirmed the results of Erlenhov et al. [26]. The main reasons for adopting bots are related to improving developer feedback, automating routine tasks, and ensuring high-quality standards. Interestingly, developers also report benefits related to interpersonal relationships. According to the surveyed developers, negative feedback in an automatic bot report feels less rude or intimidating than if a human would provide the same feedback. They also report that by providing quick and constant feedback, bots reduce the chance that a PR gets abandoned by its author.

Bots can also help to support developers unfamiliar with a software project or with specific software engineering practices and technologies. For example, Brown and Parnin [8] propose a bot to nudge students toward applying better software engineering practices. They designed a bot that provides daily updates on software development processes based on students' code contributions on GitHub. They show that such a bot can improve development practices and increase code quality and productivity.

The use of bots to automate development workflows can also result in a change in the habits of project contributors. Wessel et al. [58] investigated how activity traces change after the adoption of bots. They observed that after bot adoption, projects have more merged PRs, fewer comments, fewer rejected PRs, and faster PR rejections. Developers explain that some of these observed effects are caused by increased visibility of code quality metrics, immediate feedback, test automation, the increased confidence in the process, change in the discussion focus, and the fact that bot feedback pushes contributors to take action.

In summary, the literature suggests that developers who employ bots primarily expect **improved productivity** [26, 59]. This, however, surfaces in different ways depending on the context and the tasks the bot performs. **Automating time-consuming or tedious tasks** and **collecting dispersed information** (i.e., information gathering) are some ways to improve productivity. Developers also emphasize that bots may **perform some tasks better than humans** (e.g., handling tasks 24/7 and at scale, increasing consistency, and mitigating human error).

8.2.4 Challenges of Using Development Bots

Despite the numerous benefits leveraged by using development bots, several challenges have been reported concerning the workflow automation provided by them [26, 63]. Some bots have been studied in detail, revealing the challenges and limitations of their PR interventions [7, 44, 46].

Trust Trusting a bot to act appropriately and reliably is challenging [26]. A side effect of overly relying on bots is that humans no longer question whether these bots are taking the correct actions since they assume bots to be experts in their tasks.

Therefore, developers can be caught off guard by excessive incorrect outcomes from bots [26]. A key solution to increase trust is building a reliable testing environment that allows developers to try out bots and avoid unanticipated problems.

Discoverability and Configuration To confirm the challenges caused by development bots in PR interactions, Wessel et al. [63] interviewed 21 practitioners. Their study revealed several challenges raised by bot usage, such as discoverability and configuration issues. Developers complained about the lack of contextualized actions, limited and burdensome configuration options, and technical overhead to host and deploy their own bot. Moreover, the overload of information generated by bots when interacting on PRs has appeared as the most prominent challenge.

Interruption and Noise Developers constantly struggle with interruptions and noise produced by bots [26]. For instance, Brown and Parnin [7] analyzed `tool-recommender-bot`, a bot that automatically configures a project to use an open-source static analysis tool for Java code and then submits a PR with a generic message explaining how the proposed tool works. They reported that this bot still needs to overcome problems such as notification workload. They applied `tool-recommender-bot` in real projects for evaluation purposes. Only two PRs out of 52 proposed recommendations were accepted. Peng and Ma [46] studied how developers perceive and work with `mention bot`, a reviewer recommendation bot created by Facebook. It automatically tags a potential reviewer for a PR depending on the files changed. Project maintainers with higher expertise (i.e., maintainers who contributed more frequently) in a particular file are more likely to be suggested as reviewers by the bot. The study found that `mention bot` reduced contributors' effort in identifying proper reviewers. As a negative side effect, however, developers were bothered by frequent review notifications when dealing with a heavy workload.

Wessel et al. [63] introduced a theory about how certain bot behaviors can be perceived as noisy. Indeed, many bots provide several comments when an issue or PR is opened by a contributor, with dense information and frequently overusing visual elements. Similarly, bots perform repetitive actions such as creating numerous PRs (e.g., to update the many dependencies a project can have) and leaving dozens of comments in a row (e.g., to report on test coverage each time a new commit is added to the PR). These situations can lead to information and notification overload, disrupting developers' communication. Oftentimes, the problem is not a singular bot that is too verbose, but a combination of multiple bots that are simultaneously active and, together, lead to information overload [26].

Researchers have attempted to create solutions to reduce the information overload created by bots. Wessel et al. [57] suggested creating better ways to represent the information of bots, such as clearer summaries of pull requests. Ribeiro et al. [47] implemented `FunnelBot` that integrated these suggestions. Figure 8.3 shows an example of a PR comment posted by `FunnelBot`. The comment shows (A) an introductory message, (B) a list with all groups of bot messages collapsed, and (C) one expanded example where we can see the `CodesandBox` comment.
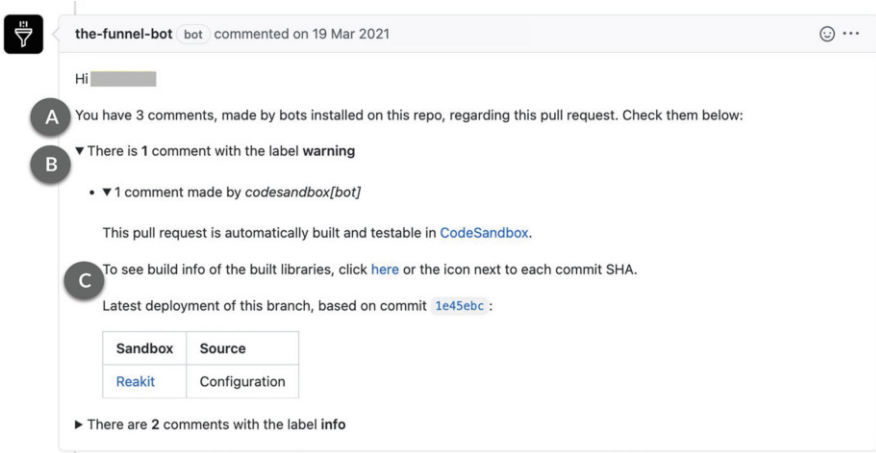
**Fig. 8.3** Example of PR comment created by `FunnelBot`

8.3 Workflow Automation Through GitHub Actions

As explained in Sect. 8.1.4, software development workflows can be automated using different techniques, including CI/CD solutions (presented in Sect. 8.1.3) and development bots (presented in Sect. 8.2). The third way is GitHub Actions, which is the focus of the current section. We explain what GitHub Actions are, how prevalent they are, and how they constitute an ecosystem of their own. We also discuss the potential challenges this novel ecosystem is confronted with.

8.3.1 What Is GitHub Actions?

The GitHub social coding platform has introduced GitHub Actions as a way to enable the specification and execution of automated workflows. It started as a beta product in 2018 providing the possibility to create Actions inside containers to augment and connect software development workflows. When the product was officially released to the public in November 2019, GitHub Actions also integrated a fully featured CI/CD service, answering the high demand of GitHub users to provide CI/CD support similar to what was already available in competing social coding platforms such as GitLab and Bitbucket [16].

Since its introduction, GitHub Actions has become the dominant CI/CD service on GitHub based on a quantitative study by Golzadeh et al. [32], including more than $90K$ GitHub repositories. Figure 8.4 provides a historical overview of CI/CD usage in those repositories, starting from the first observation of Travis usage in June 2011. Initially, GitHub repositories primarily used Travis as a CI/CD service. Over
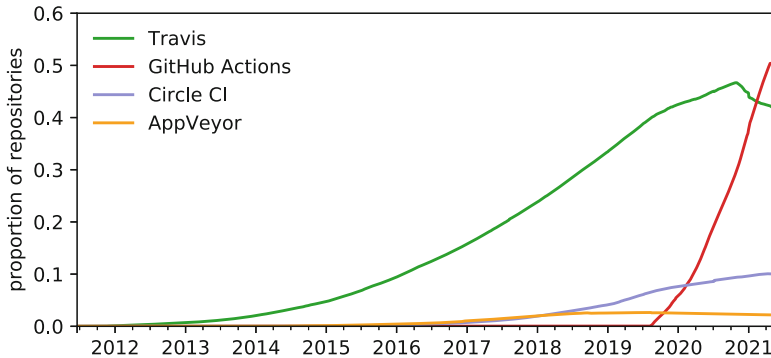
Fig. 8.4 Evolution of the proportion of GitHub repositories using a specific CI/CD solution

time, other CI/CD solutions were used, but Travis remained the dominant CI/CD. When GitHub Actions entered the CI/CD landscape, it overtook the other CI/CD solutions in popularity in less than 18 months after its introduction. Mazrae et al. [43] complemented this quantitative analysis by qualitative interviews to understand the reasons behind GitHub Actions becoming the dominant CI/CD tool in GitHub, as well as why project maintainers decided to migrate to GitHub Actions primarily. The main reported reasons were the seamless integration into GitHub, the ease of use, and great support for its reusable Actions.

GitHub Actions allows repository maintainers to automate a wide range of tasks. In addition to providing typical CI/CD services such as building code, executing test suites, and deploying new releases, GitHub Actions' tight integration with GitHub enables it to include better support of third-party tools and build support for well-known operating systems and hardware architectures, and more scalable cloud-based hardware to produce results faster. GitHub Actions also facilitates the communication between the project and external tools (such as third-party CI/CD services) and easier dependency and security monitoring and management [22].

Specifying Executable Workflows GitHub Actions is based on the so-called concept of executable *workflows* that can be defined by maintainers of GitHub repositories. The structure of a workflow is schematically presented in Fig. 8.5 and explained below.

A workflow constitutes a configurable automated process that is defined by a YAML file added to the `.github/workflows` directory of the GitHub repository. A workflow can be executed based on events specified in the workflow description that act as a trigger for running the workflow. Examples of such triggers are commits, issues, PRs, comments, schedules, or even manual invocation [11]. The example workflow in Listing 8.1 (lines 3–6) defines three possible triggers: upon committing (`push:`) or receiving a PR (`pull_request:`), or based on a specified time schedule (`cron: "0 6 * * 1"`).
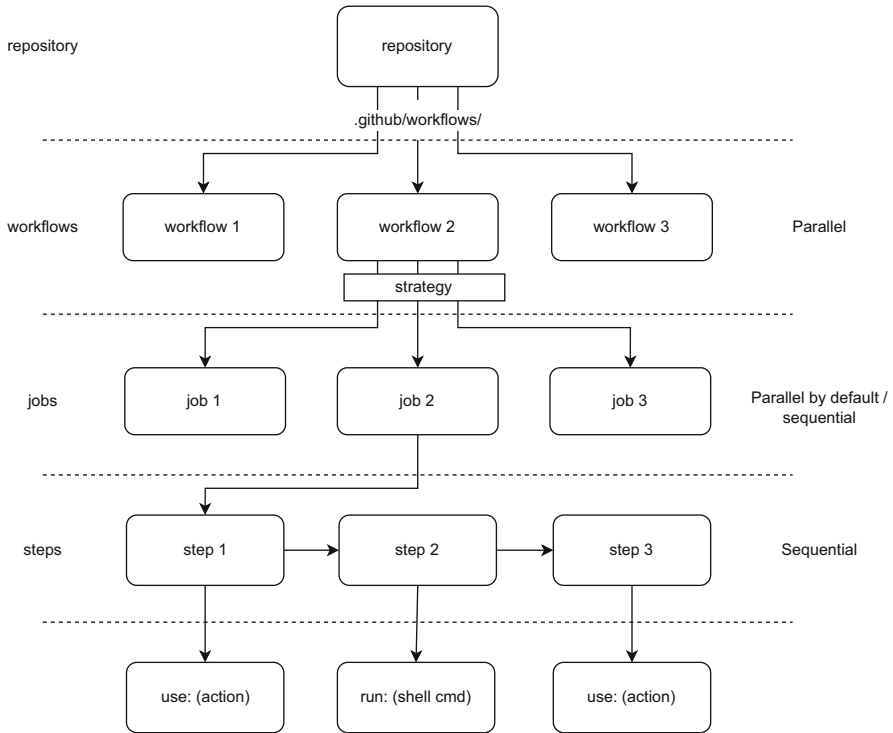
Fig. 8.5 Schematic representation of the structure of a GitHub workflow description

A workflow typically runs one *job* in some virtual environment that is created to execute the job (e.g., an instance of some specific version of Ubuntu, macOS, or Microsoft Windows). A workflow can also execute multiple jobs, in parallel (by default) or sequentially. Workflows can define a *matrix strategy* to automatically create and run parallel jobs based on the combination of variable values defined by the matrix. This is, for example, useful if one would like to build and test source code in multiple versions of a programming language and/or on multiple operating systems. In the example of Listing 8.1, the matrix strategy (lines 10–13) specifies that the job will be run on five different versions of Python for two different operating systems.

To run a workflow specified in a GitHub repository, developers can use the infrastructure provided by GitHub, or rely on self-hosted runners if more specific hardware or operating systems are needed. Each job is composed of a series of *steps* that specify the tasks to be executed sequentially by the job. These steps can be simple shell commands to be run within the virtual environment (such as lines 22–24 in Listing 8.1). Alternatively, steps can use and execute predefined reusable *Actions*, which will be discussed below.

Listing 8.1 Example of a YAML workflow file

```yaml
1 name: Test project
2 on:
3   push:
4   pull_request:
5   schedule:
6     - cron: "0 6 * * 1"
7   jobs:
8     build-and-test:
9       strategy:
10        matrix:
11          os: [ubuntu-22.04, windows-latest]
12          python: ["3.6", "3.7", "3.8", "3.9", "3.10"]
13      runs-on: ${{ matrix.os }}
14      steps:
15        - uses: actions/checkout@v2
16        - name: Set up Python
17          uses: actions/setup-python@v2
18          with:
19            python-version: ${{ matrix.python }}
20        - name: Install dependencies
21          run: |
22            pip install -r requirements.txt
23            pip install pytest
24        - name: Execute tests
25          run: pytest
```

Reusable Actions Actions provide a reuse mechanism for GitHub workflow maintainers to avoid reinventing the wheel when automating repetitive activities [12]. Rather than manually defining the sequence of commands to execute as part of a step (such as lines 22–24 in Listing 8.1), it suffices to use a specific (version of a) reusable Action. For example, line 16 in Listing 8.1 (re)uses version 2 of `actions/checkout`, and line 18 (re)uses version 2 of `actions/setup-python`. Actions are themselves developed through GitHub repositories.[4]

Workflows can reuse any Action shared in a public repository. To facilitate finding such Actions, the GitHub *Marketplace* provides an interface for providers to promote their Actions and for consumers to easily search for suitable Actions.[5] The number of Actions promoted on the Marketplace has been growing exponentially. By December 2022, the Marketplace listed over 16K reusable Actions falling

---

[4] The GitHub repositories for the Actions reused in Listing 8.1 are https://github.com/actions/checkout and https://github.com/actions/setup-python.

[5] See https://github.com/marketplace. In addition to Actions, the marketplace also promotes Apps, which are applications that can contain multiple scripts or an entire application.

under 19 different categories. These categories contain a wide diversity of Actions, covering tasks such as setting up a specific programming language environment, publishing a release on a package registry, running tests, or checking the code quality [22].

8.3.2   Empirical Studies on GitHub Actions

Given that GitHub Actions was publicly introduced in 2019, and despite the fact that GitHub Actions has become the dominating CI/CD solution on GitHub (according to Golzadeh et al. [32]), very few empirical studies have focused on GitHub Actions at the time of writing this chapter.

An early quantitative study by Kinsman et al. [40] in 2021 reported that in a dataset of 416,266 GitHub repositories, only as little as 3190 repositories (i.e., less than 1%) had been using GitHub Actions. In 2022, Wessel et al. [62] studied a dataset composed of the 5000 most-starred GitHub repositories and observed that 1489 projects (i.e., 29.8%) had been using GitHub Actions. Also in 2022, Decan et al. [22] reported on a dataset of 67,870 active GitHub repositories in which 29,778 repositories (i.e., 43.9%) had been using GitHub Actions. These quantitative results reveal that GitHub Actions is prevalent in software development repositories on GitHub. To complement these quantitative findings, in 2023, Saroar and Nayebi [48] carried out a survey with 90 GitHub developers about the best practices and perception in using and maintaining GitHub Actions.

Table 8.1 reports the top six programming languages that most frequently coincide with GitHub Actions usage according to Decan et al. [22]. They observed that some programming languages are more likely to coincide with GitHub Actions usage than others: TypeScript and Go have a higher proportion of repositories resorting to GitHub Actions usage (58.5% and 57.2%, respectively) compared to JavaScript (34.9%). It is worth noting that the percentages of repositories using GitHub Actions are reported with respect to the language itself. For example, the number of Python repositories using GitHub Actions is 1.52 times higher

Table 8.1 Top six languages with the highest proportion of GitHub repositories using GitHub Actions according to [22]

	GitHub repositories	
programming language	all repositories	using GitHub Actions
JavaScript	13,542 (19.6%)	4730 (34.9%)
Python	12,319 (17.8%)	5654 (45.9%)
TypeScript	6362 (9.2%)	3722 (58.5%)
Java	6105 (8.8%)	2390 (39.2%)
C++	5701 (8.2%)	2331 (40.9%)
Go	4988 (7.2%)	2854 (57.2%)

(5654) than the TypeScript repositories using GitHub Actions (3722). One can also observe the number of repositories for a specific language and its proportion to all the repositories in the dataset. For example, the 13,542 JavaScript repositories correspond to 19.6% of all the repositories in the dataset.

The same study also analyzed which event types are mostly used for triggering workflows, reporting that push: and pull_request: are the most frequent events triggering workflows, both used by more than half (63.4% and 56.3%, respectively) of all considered GitHub repositories relying on workflows. This is not surprising since commits and PRs are the most important activities in collaborative coding on GitHub. The most frequently used Action is actions/checkout (used by 35.5% of all steps and 97.8% of all repositories). Other frequently used Actions are related to the deployment of a specific programming language environment (e.g., setup-node, setup-python, setup-java). Overall, 24.2% of all steps use an Action of the form setup-*.

Finally, they observe that it is common practice to depend on reusable Actions, given that nearly all repositories (>99%) that use workflows have at least one step referring to an Action. More than half of the steps in all analyzed workflows (51.1%) use an Action. However, this reuse is concentrated toward a limited set of Actions. For example, the Actions that are officially provided by GitHub (i.e., those actions of the form actions/*) account for 71.7% of all steps that reuse an Action. In addition to this, the Actions being reused tend to be concentrated in a few categories. Table 8.2 provides the top five categories of Actions used by GitHub repositories, as reported by two independently conducted empirical studies [22, 62]. Most of the reused actions belong to the "Utilities" and "Continuous Integration" categories, followed by "Deployment." This suggests that GitHub Actions is being used mostly to automate the same kinds of activities as what traditional CI/CD tools are being used for.

Wessel et al. [62] statistically studied the impact of using workflows on different aspects of software development like PRs, commit frequency, and issue resolution efficiency. By comparing the activities in projects using GitHub Actions, during one full year before the use of GitHub Actions in the project and one full year after its usage, they used the technique of regression discontinuity analysis to provide statistical evidence and showed that after adding GitHub Actions to projects, there tend to be fewer accepted PRs, with more discussion comments and fewer commits,

Table 8.2 Top five most frequent Action categories according to [22, 62]

| Action category	% reported by [62]	% reported by [22]
utility	24.9%	23.9%
continuous integration	24.7%	17.3%
deployment	9.6%	7.2%
publishing	8.4%	6.9%
code quality	7.7%	6.1%

which take more time to merge. On the other hand, there are more rejected PRs, which contain fewer comments and more commits.

Wessel et al. [62] studied discussions between developers about the usage of GitHub Actions in their software projects. Out of the 5000 analyzed GitHub repositories, only 897 (18%) had the Discussions feature enabled at the time of data collection, and 830 of those (17%) contained at least one discussion thread. Focusing on this subset of repositories, they filtered the discussions containing the string "GitHub Actions," resulting in 573 posts in 458 distinct threads of 148 repositories. The most discussed material about GitHub Actions, found in 28.8% of all considered posts, was *the need for help* with GitHub Actions. This reveals that developers actively sought to learn more about how to use workflows effectively. A second popular category of discussion in the context of GitHub Actions, found in 19.0% of all considered posts, was *error messages or debug messages*. Developers were trying to solve issues related to using workflows and applications invoked via these workflows, such as linters or code review bots. A third popular category, accounting for 14.6% of all considered posts, involved discussions around *reusing Actions*. This is expected, given that Actions are a relatively new concept that many developers are not familiar with.

8.3.3 The GitHub Actions Ecosystem

As mentioned in Sect. 8.1.4, GitHub Actions is part of the larger workflow automation ecosystem of GitHub that also includes bots and CI/CD solutions for automating development workflows in collaborative software projects. Decan et al. [22] suggested that GitHub Actions can and should be considered as a new emerging ecosystem in its own right. Indeed, the GitHub Actions technology exhibits many similarities with more traditional software packaging ecosystems such as npm (for JavaScript), Cargo (for Rust), Maven (for Java), or PyPI (for Python), to name but a few. Just as software development repositories on GitHub tend to depend on external packages distributed through the above package managers—mainly to avoid the effort-intensive and error-prone practices of copy-paste reuse—the same is valid for development workflows. Maintainers of GitHub repositories can specify their workflows to directly depend on reusable Actions. As such, GitHub Actions forms a kind of *dependency network* that bears many similarities with the ones of software packaging ecosystems [18]. The parallel with packaging ecosystems is quite obvious: automated workflows, as software clients, express *dependencies* toward Actions (being the equivalent of reusable *packages*) that can exist in different *versions* or *releases*. Section 8.3.2 reported on quantitative evidence that resorting to reusable Actions in workflows has become a common practice.

Continuous Growth Decan et al. [18] carried out a quantitative empirical analysis of the similarities and differences in the evolution of the dependency networks for seven different packaging ecosystems of varying sizes and ages, including Cargo,
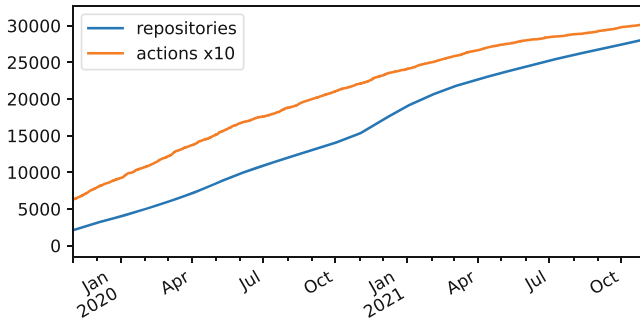
Fig. 8.6 Evolution of the number of GitHub repositories using workflows (blue line) and the number of Actions used by these repositories (orange line, scaled by a factor of 10 for ease of comparison)

CPAN, CRAN, npm, NuGet, Packagist, and RubyGems. They observed that these dependency networks tend to grow over time, both in size and in number of package updates. While the vast majority of packages depend on other packages, only a small proportion of these packages account for most of the reuse (i.e., they are targeted by most of the reverse dependencies). Decan et al. [22] conducted a quantitative analysis of GitHub Actions and observed similar characteristics for the GitHub Actions dependency network: nearly all the repositories with GitHub Actions workflows depend on reusable Actions, and most of the reuse is concentrated in a limited number of Actions. They analyzed the evolution of the number of repositories using GitHub Actions workflows and the number of Actions being used by these repositories. Figure 8.6 shows this evolution for the period 2020–2021, revealing a continuous growth of the GitHub Actions ecosystem, in terms of the number of *consumers* (repositories using GitHub Actions workflows) as well as *producers* (Actions being reused by GitHub repositories).

8.3.4   Challenges of the GitHub Actions Ecosystem

While packaging ecosystems are extremely useful for their respective communities of software developers, they have been shown to face numerous challenges related to dependency management [21, 41, 52], outdatedness [19], security vulnerabilities [20, 69], breaking changes [17, 24], deprecation [13], and abandonment of package maintainers [4, 14]. We posit that GitHub Actions will suffer (and likely suffers already) from similar issues.

Outdatedness Software developers are continuously confronted with the difficult choice of whether, when, and how to keep their dependencies up to date. On the one hand, updating a dependency to a more recent version enables them to benefit from

the latest bug and security fixes. On the other hand, doing so exposes the dependent project to an increased risk of breaking changes, as well as to new bugs or security issues that may not even have been discovered yet.

The concept of *technical lag* was proposed to measure the extent to which a software project has outdated dependencies [33]. This lag can be quantified along different dimensions: as a function of time (how long has a dependency been outdated), version (how many versions is a dependency behind), stability (how many known bugs could have been fixed by updating the dependency), and security (how many security vulnerabilities could have been addressed by updating the dependency). Zerouali et al. [67] formalized this concept in a measurement framework that can be applied at the level of packaging ecosystems. In particular, they analyzed the technical lag of the npm packaging ecosystem, observing that around 26% of the dependencies expressed by npm packages are outdated and that half of these outdated dependencies target a version that is 270+ days older than the newer one. Other researchers have also applied technical lag to quantify outdatedness in software package dependency networks [19, 54]. The technical lag framework was also applied to the ecosystem of Docker containers distributed through Docker Hub [68]. Chapter 9 provides more details on this matter.

In a similar vein, applying the technical lag framework to the GitHub Actions ecosystem would allow workflow developers to detect and quantify the presence of outdated Actions in workflows and help in updating them. It is important to do so since, despite the recency of GitHub Actions, according to Decan et al. [22], at least 16% of the dependencies in workflows are targeting an old major version of an Action.

Adherence to Semantic Versioning *Semantic Versioning* (abbreviated to *SemVer* hereafter) is another mechanism that has been proposed to assist software developers with the delicate trade-off between benefiting from security or bug fixes and being exposed to breaking changes in dependencies. SemVer introduces a set of simple rules that suggest how to assign version numbers in packages to inform developers of dependent software about potentially breaking changes. In a nutshell, SemVer proposes a three-component version scheme major.minor.patch to specify the type of changes that have been made in a new package release. Many software packaging ecosystems (such as npm, Cargo, and Packagist) are mostly SemVer compliant, in that most of their package producers adhere to the SemVer convention [17]. Backward-incompatible changes are signalled by an update of the major component, while supposedly compatible changes come with an update of either the minor or patch component. This allows dependent packages to use so-called dependency constraints to define the range of acceptable versions for a dependency (e.g., it would be safe to accept all dependency updates within the same major version range if the dependency is trusted to be SemVer compliant).

Maintainers of GitHub Actions workflows are exposed to a similar risk of incompatible changes in the Actions they use, whether these are logical changes (affecting the behavior of the Actions) or structural changes (affecting the parameters or return values). Therefore, knowing whether an Action adheres to SemVer is helpful

for maintainers of workflows depending on these actions, since they can assume minor and patch updates to be backward compatible and, therefore, free of breaking changes. GitHub recommends reusing Actions in workflows by specifying only the major component of the Action's version, allowing workflow maintainers to receive critical fixes and security patches while maintaining compatibility. However, little is known about the actual versioning practices followed by producers and consumers of Actions. Preliminary results suggest that GitHub's recommendation is widely followed since nearly 90% of the version tags used to refer to an Action include only a major component [22]. However, unlike package managers, GitHub Actions offers no support for dependency constraints, implying that producers of Actions are required to move these major version tags each time a new version of the Action is released. Unless automated, this requirement introduces an additional burden on the Action producers [22] and calls for a more profound analysis of the kind of changes made in Action updates and of the versioning practices they follow.

Security Vulnerabilities Another issue is that any software project is subject to security vulnerabilities. Package dependency networks have made the attack surface of such vulnerabilities several orders of magnitude higher due to the widespread dependence on reusable software libraries that tend to have deep transitive dependency chains [2, 20, 25, 69]. For example, through a study of 2.8K vulnerabilities in the npm and RubyGems packaging ecosystems, Zerouali et al. [66] found around 40% of the packages to be exposed to a vulnerability due to their (direct or transitive) dependencies, and it often took months to fix them. They also observed that a single vulnerable package could expose up to two-thirds of all the packages depending on it.

We see no reason why the GitHub Actions ecosystem would be immune to this phenomenon. Indeed, relying on reusable Actions from third-party repositories or even from the Marketplace further increases the vulnerability attack surface. Since a job in a workflow executes its commands within a runner shared with other jobs from the same workflow, individual jobs in the workflow can compromise other jobs they interact with. For example, a job could query the environment variables used by a later job, write files to a shared directory that a later job processes, or even more directly interact with the Docker socket and inspect other running containers and execute commands in them.[6] Multiple examples of security issues in workflows have been reported, sometimes with potentially disastrous consequences, such as manipulating pull requests to steal arbitrary secrets,[7] injecting arbitrary code with workflow commands,[8] or bypassing code reviews to push unreviewed code.[9] Unfortunately, we are not aware of any publicly available quantitative analysis on the impact of reusable Actions on security vulnerabilities in software projects.

---

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

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

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

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

This shows that there is an urgent need for further research as well as appropriate tooling to support developers of reusable Actions and workflows in assessing and hardening their security. A first step in this direction is GitHub's built-in dependency monitoring service `Dependabot`, which has started to support GitHub Actions workflows in January 2022 and reusable Actions in August 2022.[10]

Abandonment and Deprecation Another important challenge that packaging ecosystems face is the risk of packages becoming unmaintained or deprecated [13] when some or all of their core contributors have abandoned the package development [4, 14, 39]. If this happens, the packages may become inactive, implying that bugs and security vulnerabilities will no longer be fixed. This will propagate to dependent packages that rely on such packages. Cogo et al. [13] have studied the phenomenon of package deprecation in the npm packaging ecosystem, observing that 3.2% of all releases are deprecated, 3.7% of the packages have at least one deprecated release, and 66% of the packages with deprecated releases are fully deprecated. Constantinou et al. [14] studied the phenomena of developer abandonment in the RubyGems and npm packaging ecosystems to determine the characteristics that lead to a higher probability of abandoning the ecosystem. Developers were found to present such a higher risk if they do not engage in discussions with other developers, do not have strong social and technical activity intensity, communicate or commit less frequently, and do not participate in both technical and social activities for long periods of time. Avelino et al. [4] carried out a mixed-methods study to investigate project abandonment in popular GitHub projects, revealing that some projects recovered from the abandonment of key developers because they were taken over by new core maintainers that were aware of the project abandonment risks and had a clear incentive for the project to survive.

Since Actions are reusable software components being developed in GitHub repositories, the GitHub Actions ecosystem is likely to suffer from this risk of abandoning developers and the presence of unmaintained or obsolete Actions. This calls for studies to quantify this phenomenon and mechanisms to avoid abandonment or to provide solutions to overcome the negative effects of such abandonment. Examples of such solutions could be finding the right replacement for abandoning developers in Action repositories or suggesting consumers of unmaintained Actions to migrate to alternative Actions.

Beyond GitHub Actions The exposure of GitHub Actions to the well-known issues that packaging ecosystems face is all the more worrying because they are not limited to the GitHub Actions ecosystem but may also affect other packaging ecosystems. Conversely, the GitHub Actions ecosystem may be affected by issues coming from packaging ecosystems. This situation is depicted in Fig. 8.7: GitHub hosts the development repositories of many software projects distributed in packaging ecosystems. These development repositories may define automated

---

[10] https://github.blog/2022-08-09-dependabot-now-alerts-for-vulnerable-github-actions/.
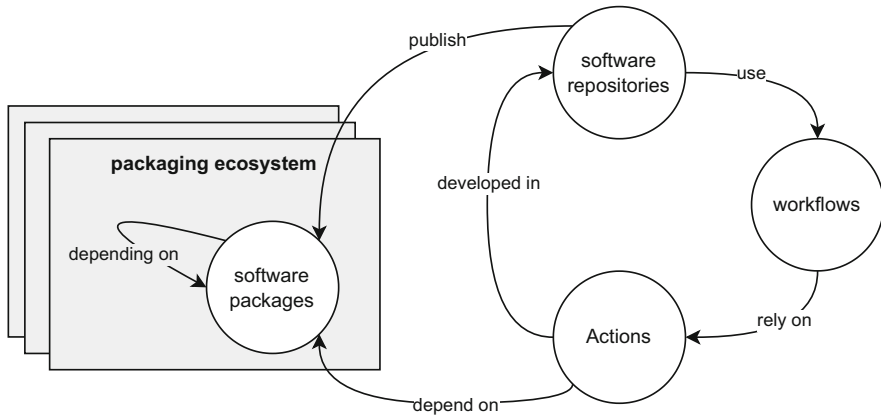
Fig. 8.7 Interweaving of the GitHub Actions ecosystem and software packaging ecosystems

workflows relying on reusable Actions. The Actions themselves are also developed in (and directly accessed through) GitHub repositories. Since Actions are software components developed in some programming language (mostly in TypeScript currently), they may depend on reusable packages or libraries distributed in package registries such as npm.

This potentially strong interconnection between GitHub Actions and packaging ecosystems is not without practical consequences given the issues that these ecosystems may face. Instead of being mostly limited to their *own* ecosystem, issues affecting either packages or Actions may cross the boundaries and propagate to the other software ecosystems they are interwoven with.

Consider, for example, a reusable Action affected by a security vulnerability. To start with, this vulnerability may compromise all the workflows relying on the affected Action. Next, it may also compromise the development repositories in which these workflows are executed. By extension, it may also affect all the software projects developed in these repositories. In turn, these projects may affect all the dependent packages that use them and so on. For example, the `action-download-artifact` Action, used by several thousands of repositories, was found to expose workflows using it to code injection attacks.[11] Conversely, Actions may depend on vulnerable packages distributed in a packaging ecosystem such as npm. As a consequence, issues affecting these packages may propagate to the Actions using them and may in turn propagate to the workflows and development repositories relying on these Actions.

In summary, many of the issues that software packaging ecosystems have been shown to face also apply directly or indirectly to the GitHub Actions ecosystem.

---

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

Even worse, given that both kinds of ecosystems are tightly interwoven, issues in either ecosystem can and will propagate across ecosystem boundaries, which may lead to a significantly increased exposure to vulnerabilities and other socio-technical health issues. This raises the urgent need to conduct empirical research for understanding the extent of these issues, analyze their impact and propagation, and provide tool support for helping repository, package, and workflow maintainers.

8.4 Discussion

This chapter focused on the emerging ecosystems of development workflow automation in the GitHub social coding platform, consisting of the socio-technical interaction with bots (automated software development agents), and the workflow automation offered through GitHub Actions. Combined together, GitHub's socio-technical ecosystem comprises human contributors, bots, workflows and reusable actions, GitHub Apps,[12] and all of the GitHub repositories in which these technologies are being developed and used. It also comprises external CI/CD services or other development automation tools that may be used by these GitHub repositories. In addition to this, there is a tight interweaving with software packaging ecosystems, since software packages may be developed using bots and GitHub Actions, and the development of bots and Actions may depend on software packages. We have argued that the intricate combination of workflow automation solutions constitutes an important and increasing risk that exposes the involved repositories—and, by extension, the software products they generate or that depend on them—to vulnerabilities and other socio-technical issues. Similar issues are likely to apply to other social coding platforms (e.g., GitLab and Bitbucket) for the same reasons as in GitHub, even though the workflow automation solutions and technologies in those platforms may be different.

We also argued that bots play an important role in the *social* fabric of the GitHub ecosystem, since bots interact and communicate with human contributors using a similar interface as the one used by humans (e.g., posting and reacting to comments on issues, PRs, code reviews, and commits in repositories). Actions, on the other hand, are more commonly used to automate *technical* tasks such as executing test suites and deploying packages. This was quantitatively observed by Decan et al. in [22].

However, the boundaries between the bot ecosystem and the GitHub Actions ecosystem are becoming more and more diffuse. For instance, nothing prevents bots from directly using the functionality offered by Actions (e.g., a bot could trigger the execution of a workflow using Actions that run test suites). Similarly, an Action may instruct a bot to interact with developers and users (e.g., a code coverage Action may report its results through some GitHub badge, issue, or PR comment).

---

[12] https://docs.github.com/en/developers/apps.

Existing workflow automation solutions were already offered through a wide variety of channels for GitHub, for example, through CI/CD services, external bots, dedicated web interfaces, or GitHub Apps. The introduction of GitHub Actions has further increased the overlap between the possible automation services. For instance, some automation services that used to be offered through bots or GitHub Apps have now become available as Actions as well. An example is the GitHub App `the-welcome-bot` for welcoming newcomers, a task for which more recently a GitHub Action `wow-actions/welcome` has become available. Two other examples are the `renovate` dependency update service and the `codecov` code coverage analysis that used to be available through web services and GitHub Apps, and more recently codecov has become offered as a GitHub Action as well. Going one step further, `dependabot`, which used to be an independent bot service, has now become fully integrated into the GitHub platform.

All these examples illustrate that bots, Apps, Actions, and external services will continue to coexist side by side as part of the development workflow ecosystem. It is yet unclear to which extent GitHub repositories are using a combination of workflow automation solutions, or to which extent they tend to migrate from one solution to another. Hence, empirical studies that shed a deeper insight into this rapidly expanding ecosystem are urgently needed.

References

1. Abdellatif, A., Wessel, M., Steinmacher, I., Gerosa, M.A., Shihab, E.: BotHunter: an approach to detect software bots in GitHub. In: International Conference on Mining Software Repositories (MSR), pp. 6–17. IEEE Computer Society (2022). https://doi.org/10.1145/3524842.3527959

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

3. Arora, R., Goel, S., Mittal, R.: Supporting collaborative software development over GitHub. Softwa. Pract. Exper. **47** (2016). https://doi.org/10.1002/spe.2468

4. Avelino, G., Constantinou, E., Valente, M.T., Serebrenik, A.: On the abandonment and survival of open source projects: an empirical investigation. In: International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–12 (2019). https://doi.org/10.1109/ESEM.2019.8870181

5. Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., et al.: Manifesto for agile software development. Tech. rep., Snowbird, UT (2001)

6. Beller, M., Gousios, G., Zaidman, A.: Oops, my tests broke the build: an explorative analysis of Travis CI with GitHub. In: International Conference on Mining Software Repositories (MSR), pp. 356–367. IEEE, Piscataway (2017). https://doi.org/10.1109/MSR.2017.62

7. Brown, C., Parnin, C.: Sorry to bother you: designing bots for effective recommendations. In: International Workshop on Bots in Software Engineering (BotSE). IEEE, Piscataway (2019). https://doi.org/10.1109/BotSE.2019.00021

8. Brown, C., Parnin, C.: Nudging students toward better software engineering behaviors. In: International Workshop on Bots in Software Engineering (BotSE), pp. 11–15. IEEE, Piscataway (2021). https://doi.org/10.1109/BotSE52550.2021.00010

9. Cassee, N., Kitsanelis, C., Constantinou, E., Serebrenik, A.: Human, bot or both? A study on the capabilities of classification models on mixed accounts. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 654–658. IEEE, Piscataway (2021). https://doi.org/10.1109/ICSME52107.2021.00075

10. Catolino, G., Palomba, F., Tamburri, D.A., Serebrenik, A.: Understanding community smells variability: a statistical approach. In: International Conference on Software Engineering (ICSE), pp. 77–86 (2021). https://doi.org/10.1109/ICSE-SEIS52602.2021.00017

11. Chandrasekara, C., Herath, P.: Hands-on GitHub Actions: Implement CI/CD with GitHub Action Workflows for Your Applications. Apress (2021). https://doi.org/10.1007/978-1-4842-6464-5

12. Chen, T., Zhang, Y., Chen, S., Wang, T., Wu, Y.: Let's supercharge the workflows: an empirical study of GitHub Actions. In: International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE, Piscataway (2021). https://doi.org/10.1109/QRS-C55045.2021.00163

13. Cogo, F.R., Oliva, G.A., Hassan, A.E.: Deprecation of packages and releases in software ecosystems: a case study on npm. Transactions on Software Engineering (2021). https://doi.org/10.1109/TSE.2021.3055123

14. Constantinou, E., Mens, T.: An empirical comparison of developer retention in the RubyGems and npm software ecosystems. Innovations Syst. Softw. Eng. **13**(2), 101–115 (2017). https://doi.org/10.1007/s11334-017-0303-4

15. Costa, J.M., Cataldo, M., de Souza, C.R.: The scale and evolution of coordination needs in large-scale distributed projects: implications for the future generation of collaborative tools. In: SIGCHI Conference on Human Factors in Computing Systems, pp. 3151–3160 (2011). https://doi.org/10.1145/1978942.1979409

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

17. Decan, A., Mens, T.: What do package dependencies tell us about semantic versioning? Trans. Softw. Eng. **47**(6), 1226–1240 (2021). https://doi.org/10.1109/TSE.2019.2918315

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

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

20. Decan, A., Mens, T., Constantinou, E.: On the impact of security vulnerabilities in the npm package dependency network. In: International Conference on Mining Software Repositories (MSR), pp. 181–191 (2018). https://doi.org/10.1007/s10664-022-10154-1

21. Decan, A., Mens, T., Grosjean, P.: An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empirical Softw. Eng. **24**(1), 381–416 (2019). https://doi.org/10.1007/s10664-017-9589-y

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

23. Dey, T., Mousavi, S., Ponce, E., Fry, T., Vasilescu, B., Filippova, A., Mockus, A.: Detecting and characterizing bots that commit code. In: International Conference on Mining Software Repositories (MSR), pp. 209–219. ACM (2020). https://doi.org/10.1145/3379597.3387478

24. Dietrich, J., Pearce, D., Stringer, J., Tahir, A., Blincoe, K.: Dependency versioning in the wild. In: International Conference on Mining Software Repositories (MSR), pp. 349–359. IEEE, Piscataway (2019). https://doi.org/10.1109/MSR.2019.00061

25. Düsing, J., Hermann, B.: Analyzing the direct and transitive impact of vulnerabilities onto different artifact repositories. Digit. Threats Res. Pract. (2021). https://doi.org/10.1145/3472811

26. Erlenhov, L., Neto, F.G.d.O., Leitner, P.: An empirical study of bots in software development: characteristics and challenges from a practitioner's perspective. In: Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 445–455. ACM (2020). https://doi.org/10.1145/3368089.3409680

27. Erlenhov, L., de Oliveira Neto, F.G., Scandariato, R., Leitner, P.: Current and future bots in software development. In: International Workshop on Bots in Software Engineering (BotSE), pp. 7–11. IEEE, Piscataway (2019). https://doi.org/10.1109/BotSE.2019.00009

28. Fowler, M., Foemmel, M.: Continuous Integration (original version) (2000). https://martinfowler.com/articles/originalContinuousIntegration.html. Accessed 15 Apr 2023

29. GitHub: The state of open source software 2022 (2022). octoverse.github.com. Accessed 15 Apr 2023

30. Golzadeh, M., Decan, A., Legay, D., Mens, T.: A ground-truth dataset and classification model for detecting bots in GitHub issue and PR comments. J. Syst. Softw. **175** (2021). https://doi.org/10.1016/j.jss.2021.110911

31. Golzadeh, M., Decan, A., Mens, T.: Evaluating a bot detection model on git commit messages. In: CEUR Workshop Proceedings, vol. 2912 (2021)

32. Golzadeh, M., Decan, A., Mens, T.: On the rise and fall of CI services in GitHub. In: International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, Piscataway (2021). https://doi.org/10.1109/SANER53432.2022.00084

33. Gonzalez-Barahona, J.M., Sherwood, P., Robles, G., Izquierdo, D.: Technical lag in software compilations: Measuring how outdated a software deployment is. In: IFIP International Conference on Open Source Systems, pp. 182–192. Springer, Berlin (2017). https://doi.org/10.1007/978-3-319-57735-7_17

34. Gousios, G., Pinzger, M., van Deursen, A.: An exploratory study of the pull-based software development model. In: International Conference on Software Engineering (ICSE), pp. 345–355. ACM (2014). https://doi.org/10.1145/2568225.2568260

35. Gousios, G., Storey, M.A., Bacchelli, A.: Work practices and challenges in pull-based development: the contributor's perspective. In: International Conference on Software Engineering (ICSE), pp. 285–296. ACM (2016). https://doi.org/10.1145/2884781.2884826

36. Gousios, G., Zaidman, A., Storey, M.A., van Deursen, A.: Work practices and challenges in pull-based development: the integrator's perspective. In: International Conference on Software Engineering (ICSE), pp. 358–368. IEEE, Piscataway (2015). https://doi.org/10.1109/ICSE.2015.55

37. Herbsleb, J.D.: Global software engineering: the future of socio-technical coordination. In: International Conference on Software Engineering (ISCE)—Workshop on the Future of Software Engineering, pp. 188–198. IEEE, Piscataway (2007). https://doi.org/10.1109/FOSE.2007.11

38. Holmström, H., Conchúir, E.Ó., Ågerfalk, P.J., Fitzgerald, B.: Global software development challenges: a case study on temporal, geographical and socio-cultural distance. In: International Conference on Global Software Engineering (ICGSE), pp. 3–11. IEEE, Piscataway (2006). https://doi.org/10.1109/ICGSE.2006.261210

39. Kaur, R., Kaur, K.: Insights into developers' abandonment in FLOSS projects. In: Intelligent Sustainable Systems. Lecture Notes in Networks and Systems, vol. 333. Springer, Berlin (2022). https://doi.org/10.1007/978-981-16-6309-3_69

40. Kinsman, T., Wessel, M., Gerosa, M.A., Treude, C.: How do software developers use GitHub Actions to automate their workflows? In: International Conference on Mining Software Repos-

itories (MSR), pp. 420–431. IEEE, Piscataway (2021). https://doi.org/10.1109/MSR52588.2021.00054

41. Kula, R.G., German, D.M., Ouni, A., Ishio, T., Inoue, K.: Do developers update their library dependencies? Empirical Softw. Eng. **23**(1), 384–417 (2018). https://doi.org/10.1007/s10664-017-9521-5

42. Lebeuf, C., Storey, M.A., Zagalsky, A.: Software bots. IEEE Softw. **35**(1), 18–23 (2017). https://doi.org/10.1109/MS.2017.4541027

43. Mazrae, P.R., Mens, T., Golzadeh, M., Decan, A.: On the usage, co-usage and migration of CI/CD tools: a qualitative analysis. Empirical Softw. Eng. (2023). https://doi.org/10.1007/s10664-022-10285-5

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

45. Monperrus, M.: Explainable software bot contributions: case study of automated bug fixes. In: International Workshop on Bots in Software Engineering (BotSE), pp. 12–15. IEEE, Piscataway (2019). https://doi.org/10.1109/BotSE.2019.00010

46. Peng, Z., Ma, X.: Exploring how software developers work with mention bot in GitHub. CCF Trans. Pervasive Comput. Interaction **1**(3), 190–203 (2019). https://doi.org/10.1007/s42486-019-00013-2

47. Ribeiro, E., Nascimento, R., Steinmacher, I., Xavier, L., Gerosa, M., De Paula, H., Wessel, M.: Together or apart? Investigating a mediator bot to aggregate bot's comments on pull requests. In: International Conference on Software Maintenance and Evolution—New Ideas and Emerging Results Track (ICSME-NIER). IEEE, Piscataway (2022). https://doi.org/10.1109/ICSME55016.2022.00054

48. Saroar, S.G., Nayebi, M.: Developers' perception of GitHub Actions: a survey analysis. In: International Conference on Evaluation and Assessment in Software Engineering (EASE) (2023)

49. Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., Stumm, M.: Continuous deployment at Facebook and OANDA. In: International Conference on Software Engineering Companion (ICSE), pp. 21–30. IEEE, Piscataway (2016). https://doi.org/10.1145/2889160.2889223

50. Soares, E., Sizilio, G., Santos, J., da Costa, D.A., Kulesza, U.: The effects of continuous integration on software development: a systematic literature review. Empirical Softw. Eng. **27**(3), 1–61 (2022). https://doi.org/10.1007/s10664-021-10114-1

51. Song, Y., Chaparro, O.: BEE: A tool for structuring and analyzing bug reports. In: Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 1551–1555. ACM (2020). https://doi.org/10.1145/3368089.3417928

52. Soto-Valero, C., Harrand, N., Monperrus, M., Baudry, B.: A comprehensive study of bloated dependencies in the Maven ecosystem. Empirical Softw. Eng. **26**(3), 1–44 (2021). https://doi.org/10.1007/s10664-020-09914-8

53. Storey, M.A., Zagalsky, A.: Disrupting developer productivity one bot at a time. In: International Symposium on Foundations of Software Engineering (FSE), pp. 928–931 (2016). https://doi.org/10.1145/2950290.2983989

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

55. Tsay, J., Dabbish, L., Herbsleb, J.: Influence of social and technical factors for evaluating contribution in GitHub. In: International Conference on Software Engineering (ICSE), pp. 356–366. ACM (2014). https://doi.org/10.1145/2568225.2568315

56. Wang, Z., Wang, Y., Redmiles, D.: From specialized mechanics to project butlers: the usage of bots in OSS development. IEEE Software (2022). https://doi.org/10.1109/MS.2022.3180297

57. Wessel, M., Abdellatif, A., Wiese, I., Conte, T., Shihab, E., Gerosa, M.A., Steinmacher, I.: Bots for pull requests: the good, the bad, and the promising. In: International Conference on Software Engineering (ICSE), pp. 274–286 (2022). https://doi.org/10.1145/3510003.3512765

58. Wessel, M., Serebrenik, A., Wiese, I., Steinmacher, I., Gerosa, M.A.: Effects of adopting code review bots on pull requests to OSS projects. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 1–11. IEEE, Piscataway (2020). https://doi.org/10.1109/ICSME46990.2020.00011

59. Wessel, M., Serebrenik, A., Wiese, I., Steinmacher, I., Gerosa, M.A.: What to expect from code review bots on GitHub? A survey with OSS maintainers. In: Brazilian Symposium on Software Engineering (SBES), pp. 457–462 (2020). https://doi.org/10.1145/3422392.3422459

60. Wessel, M., Serebrenik, A., Wiese, I., Steinmacher, I., Gerosa, M.A.: Quality gatekeepers: investigating the effects of code review bots on pull request activities. Empirical Softw. Eng. **27**(5), 108 (2022). https://doi.org/10.1007/s10664-022-10130-9

61. Wessel, M., de Souza, B.M., Steinmacher, I., Wiese, I.S., Polato, I., Chaves, A.P., Gerosa, M.A.: The power of bots: characterizing and understanding bots in OSS projects. Proc. ACM Hum.-Comput. Interact. **2**(CSCW) (2018). https://doi.org/10.1145/3274451

62. Wessel, M., Vargovich, J., Gerosa, M.A., Treude, C.: Github actions: the impact on the pull request process (2022). arXiv preprint arXiv:2206.14118

63. Wessel, M., Wiese, I., Steinmacher, I., Gerosa, M.A.: Don't disturb me: challenges of interacting with software bots on open source software projects. In: ACM Hum.-Comput. Interact. (CHI). ACM (2021). https://doi.org/10.1145/3476042

64. Wyrich, M., Bogner, J.: Towards an autonomous bot for automatic source code refactoring. In: International Workshop on Bots in Software Engineering (BotSE), pp. 24–28 (2019). https://doi.org/10.1109/BotSE.2019.00015

65. Wyrich, M., Ghit, R., Haller, T., Müller, C.: Bots don't mind waiting, do they? Comparing the interaction with automatically and manually created pull requests. In: International Workshop on Bots in Software Engineering (BotSE), pp. 6–10. IEEE, Piscataway (2021). https://doi.org/10.1109/BotSE52550.2021.00009

66. Zerouali, A., Mens, T., Decan, A., De Roover, C.: On the impact of security vulnerabilities in the npm and RubyGems dependency networks. Empirical Softw. Eng. **27**(5), 1–45 (2022). https://doi.org/10.1007/s10664-022-10154-1

67. Zerouali, A., Mens, T., Gonzalez-Barahona, J., Decan, A., Constantinou, E., Robles, G.: A formal framework for measuring technical lag in component repositories—and its application to npm. J. Softw. Evol. Process **31**(8) (2019). https://doi.org/10.1002/smr.2157

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

69. Zimmermann, M., Staicu, C.A., Tenny, C., Pradel, M.: Small world with high risks: a study of security threats in the npm ecosystem. In: USENIX Security Symposium, pp. 995–1010 (2019)