GASH – The GitHub Actions Smell Hunter

Matheus B. Freitas[1] **and Lincoln S. Rocha**[1]

[1]EvidenSE Research Group
Federal University of Ceará (UFC)
Fortaleza – CE – Brazil

mthsfrts@alu.ufc.br and lincoln@dc.ufc.br

Abstract. *The CI/CD pipeline configuration is a challenging and error-prone task. Its misconfiguration threatens the project's security, maintenance, and quality. Such configuration problems called "configuration smells" are patterns in the configuration that, while not necessarily incorrect, indicate potential issues that could compromise the pipeline efficiency, reliability, or security. Detecting these smells is key to managing and addressing them for maintaining high-quality and secure CI/CD workflows. This paper introduces GASH (GitHub Actions Smell Hunter), a Pythonic tool devoted to detecting configuration smells in GitHub Actions CI/CD pipelines. Our tool can detect nine configuration smells categorized into three groups: security (5), maintenance and reliability (3), and code quality (1). GASH provides features to support researchers in performing large-scale studies regarding configuration smells and practitioners in continuously analyzing their own pipelines. We evaluate GASH against a manually labeled "gold standard" based on 15 open-source projects comprising 66 CI/CD pipeline configurations. The results show that GASH performed well, achieving F_1-score greater than 0.8 for most configuration smells.*
Keywords: *CI/CD, GitHub Actions, Configuration Smells, Static Analysis*

1. Introduction

Launched as a public product by GitHub in 2019 [Kinsman et al. 2021], GitHub Actions[1] is a CI/CD platform that automates build, test, and deployment pipelines. It leverages YAML[2], a markup language, to allow users to create CI/CD pipelines as workflows within `.yml` files. According to [Wessel et al. 2023], GitHub Actions experienced impressive growth in adoption (almost 30% of popular repositories studied) when compared with the beginning (only 0.7% of popular repositories as reported in [Kinsman et al. 2021]). Key components include workflows, events, jobs, actions, and runners, detailed in Section 2.1.

Overall, the CI/CD pipeline configuration is a challenging and error-prone task [Rahman et al. 2019]. As reported by [Zhang et al. 2024], developers have faced these practical problems when deploying CI/CD workflows using GitHub Actions. Developers responsible for this task must follow certain principles and good practices to avoid misconfiguration that may threaten the project's security (e.g., storing sensitive information like tokens or passwords directly in the code) [Rahman et al. 2019], maintenance (e.g., code snippets replication in different parts of the pipeline) [Vassallo et al. 2020], and quality (e.g., extensive and hard-to-manage code blocks) [Vasilescu et al. 2015]. These

[1]`https://github.com/features/actions`
[2]`https://yaml.org/`

configuration problems are referred to as "configuration smells", i.e., patterns in the configuration that, while not necessarily incorrect, indicate a potential problem that could compromise the pipeline efficiency, reliability, or security. Detecting and addressing these smells is key for maintaining high-quality and secure CI/CD workflows.

In this paper, we present GASH (GitHub Actions Smell Hunter), a command line-based Pythonic tool to automatically detect configuration smells in CI/CD pipelines deployed on GitHub Actions. Our tool can detect nine configuration smells (e.g., Hard-Coded Secret, Admin by Default, and Lack of Error Handling) categorized into three different groups: security (5), maintenance and reliability (3), and code quality (1). GASH can analyze one or several YAML-based pipeline configurations from one or multiple repositories. These features support researchers in performing large-scale studies regarding configuration smells and practitioners in continuously analyzing their own pipelines. We evaluate GASH against a manually labeled "gold standard" based on 15 open-source projects, comprising 66 CI/CD pipeline configurations, and a total of 446 smells. Although GASH didn't get perfect scores for detecting every smell, our results show that GASH performed well, achieving F_1-score greater than 0.8 for most configuration smells.

The rest of the paper is organized as follows. Section 2 presents the background and related work. GASH tool is described in detail in Section 3. In Section 4, we present GASH's evaluation methodology, results, discussion, limitations, and threats to validity. Section 5 concludes the paper and points to future research directions.

2. Background and Related Work

2.1. GitHub Actions Workflows

In GitHub Actions, workflows are custom automated processes defined in YAML files within the `.github/workflows` directory of a repository. They are triggered by events in a repository (e.g., pull request, issue creation, commit push, or a scheduled event) and can have multiple jobs. Jobs are units of work that execute in parallel or sequentially within a workflow. Each job runs on a runner and consists of several steps. Each step is either a shell script that will be executed or an action that will be run. Steps are executed sequentially and are interdependent. Since all steps in a job run on the same runner, they can share data, allowing one step to use the output or results from another. Actions are individual tasks that can be combined to create jobs. They can be custom scripts or reusable actions from the GitHub Marketplace. Finally, runners are servers that run the jobs in workflows. To ease the process, GitHub provides hosted runners such as Linux, Windows, and macOS virtual machines, but users can self-host their own.

```yaml
name: learn-github-actions
run-name: ${{github.actor}} is learning GitHub Actions
on: [push]
jobs:
  check-bats-version:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: '20'
      - run: npm install -g bats
```

```
13      - run: bats -v
```

Listing 1. Example of GitHub Actions Workflow Description in YAML File.

Listing 1 shows a GitHub Actions workflow written in YAML. In line 1, the field `name` of the workflow (i.e., how it will appear in the "Actions" tab of the repository) is optional. If the `name` is omitted, the name of the `.yml` file will be used instead. Next, in line 2, the field `run-name` is the name for workflow runs generated from the workflow, which will appear in the list of workflow runs on your repository's "Actions" tab. This example uses a GitHub expression `${{github.actor}}` to display the username of the actor that triggered the workflow run. The field `run-name` is also optional. In line 3, the field `on` specifies the trigger for this workflow. In this case, it is the push event, so a workflow run is triggered every time someone pushes a change to the repository or merges a pull request. The field `jobs` (line 4) groups all the jobs that run in the current workflow. Right after, `check-bats-version` defines the name of the first (and only one) workflow's job. The property `runs-on` configures the job to run on the latest version of an Ubuntu Linux runner. The property `steps` groups all the steps that run in the job. Each item nested under `steps` section is a separate action or shell script. The property `uses` specifies that the step will run an action. First, the action `actions/checkout` (version 4) is executed. It checks out the repository onto the runner, allowing you to run scripts or other actions against your code (e.g., perform static analysis or run automated tests). Next, the action `actions/setup-node` (version 4) is executed to install version 20 of Node.js, a cross-platform JavaScript runtime environment. Finally, using the property `run` two shell commands are executed: `rpm install -g bats` to install the `bats` software testing package on the runner and `bats -v` to display the `bats` version.

2.2. Configuration Smells in CI/CD Pipelines

Table 1 summarizes the nine configuration smells our tool can automatically identify. These smells were selected from [Rahman et al. 2019] and [Vassallo et al. 2020] list of smells. We categorize the configuration smells into three groups – security smells (SEC), maintenance and reliability smells (MRS), and code quality smells (CQS) – and assign vulnerability levels (Critical, Medium, or Low) for each smell we consider. The final level of vulnerability was reached based on three expert evaluations. First, each expert assigned values for each configuration smell level of vulnerability. Finally, they discussed among themselves and reached a consensual level of vulnerability for each smell.

Security in CI/CD pipelines is important to prevent unauthorized access, data breaches, and other vulnerabilities that can compromise the integrity of the software development lifecycle. In the context of GitHub Actions, security smells are patterns that indicate potential security risks. If not addressed, these smells can lead to vulnerabilities in the CI/CD process and software deployment and, consequently, risks to the end user. The **Hard-Coded Secret** smell (see Table 1) is a major security risk once this practice can lead to unauthorized access and data breaches if the secrets are exposed. The occurrence of **Unsecured Protocol** smells in CI/CD pipelines, such as HTTP use instead of HTTPS, can expose sensitive data to interception and man-in-the-middle attacks. The presence **Untrusted Dependencies** smell in the pipeline configuration can introduce vulnerabilities through third-party code, making the system susceptible to attacks. **Admin by Default** smell can increase the risk of unauthorized access and data breaches due to the extensive

Table 1. Summary of Configuration Smells in CI/CD Pipelines.

Group	Name	Vulnerability	Description
SES	Hard-Coded Secret	Critical	Storing sensitive information like tokens or passwords directly in the code.
	Unsecured Protocol	Critical	Lack of secure communication protocols.
	Untrusted Dependencies	Critical	Including dependencies from unverified or untrusted sources.
	Admin by Default	Critical	Assigning high privileges by default.
	Remote Triggers	Critical	Allowing remote triggers without proper configuration.
MRS	Code Replica	Medium	Replicating code snippets in different parts of the pipeline.
	Lack of Error Handling	Critical	Absence of robust error checks and handling mechanisms.
	Misconfigurations	Medium	Incorrect configurations in the workflow that cause execution failures or security vulnerabilities.
CQS	Long Code Blocks	Medium	Extensive blocks of code at the workflow, job, or step levels.

control these privileges grant (permissions are used in actions to provide specific privileges for the pipeline). Finally, allowing **Remote Triggers** without proper configuration can be exploited to execute unauthorized actions, compromising the pipeline integrity.

Replicating code snippets across different parts of the pipeline configuration (see **Code Replica** smell in Table 1) can lead to maintenance challenges and an increased risk of inconsistencies and errors. Identifying and refactoring replicated code is important for ensuring the maintainability and reliability of the CI/CD pipeline. Proper error handling is key to ensure that CI/CD pipelines can manage failures gracefully and provide useful feedback for troubleshooting. Inadequate error handling, such as those represented by the **Lack of Error Handling** smell, can result in undetected failures, prolonged debugging sessions, and unreliable pipeline performance. Finally, the presence of **Misconfigurations** smell in GitHub Actions workflows can lead to several issues (e.g., failed executions, security vulnerabilities, and maintenance challenges). Identifying and fixing these misconfigurations is mandatory to ensure the pipeline's robustness and reliability.

In GitHub Actions workflows, long code blocks (see **Long Code Blocks** smell in Table 1) can hinder maintenance and reduce code quality. Although they do not directly compromise pipeline security, they impact the readability and manageability of the workflow. Thus, it is important to consider splitting large jobs into multiple workflows and breaking down steps with excessive commands into smaller units. By doing so, not only the readability is improved but also debugging and long-term maintenance.

2.3. Related Work

In this section, we describe the related work. We start with the [Vassallo et al. 2020] work that inspired us to develop the GASH tool. [Vassallo et al. 2020] developed a linter capable of automatically identifying four different smells in pipeline configuration files. They evaluate their tool through a long-term (over 6 months) and large-scale (targeting 5,312 open-source projects hosted on GitLab platform) study. Their results show that developers are aware of configuration smells and their tool achieves a precision of 87% and a recall of 94%. They also conclude that smells are prevalent once 31% of projects with long configurations are affected by at least one smell. However, the proposed linter is limited to projects hosted on GitLab platform, considers only four smells, and does not provide dedicated support to perform studies with historical analyses, unlike GASH.

[Rahman et al. 2019] performed a study in the context of infrastructure as code (IaC) to address the problem of security smells in configuration scripts. They argue that practitioners tend to introduce security smells when creating configuration scripts for IaC. Thus, they performed a qualitative analysis on 1,726 IaC scripts to identify seven security smells. Next, they built a tool to detect the occurrence of each smell in 15,232 IaC scripts from 293 open-source repositories. They found that hard-coded passwords occur the most and take more than 90 months to be fixed. However, the proposed tool does not provide support to analyze CI/CD pipeline configuration, which is the primary goal of GASH.

3. The GASH Tool

3.1. Overview

GASH (GitHub Actions Smell Hunter) is a Python-based tool primarily devoted to identifying configuration smells in CI/CD pipelines deployed on GitHub Actions. It can parser YAML files describing CI/CD workflows looking at identifying the presence of configuration smells. Overall, GASH outputs a report containing which smells were found and their occurrences in the YAML file. Our tool detects all nine smells described in Table 1.

GASH can be used in two distinct ways. The first one considers the analysis of YAML files stored in the machine's local file system. Thus, the user can provide a path for a single file or a directory containing several YAML files. After performing the analysis, GASH outputs a report for each YAML containing the smells found and their location in the file. This way is more useful for practitioners who intend to assess the current pipeline configurations of a single project. The second way focuses on mining software repositories hosted on GitHub. In this case, the user must provide a repository URL or CSV file containing a list of URLs pointing to the repositories to be analyzed. Next, GASH downloads each repository, selects and inspects all commits that touch a YAML file, and analyzes each YAML file's version linked with the selected commits. Finally, GASH outputs a CSV file report containing a summary of information regarding the commit (e.g., committer name, committer date, and whether the commit is linked with an issue) and smells found (e.g., type and occurrences). This way is more useful to support researchers in performing pipeline history analysis, including multiple projects.

Finally, GASH follows well-known software engineering principles such as Test-Driven Development (TDD) and Domain-Driven Design (DDD). The Prisma[3] project, an ORM tool for Node.js and TypeScript with +31k stars, used by +462k projects, and that encompasses 16 pipelines, was used as the gold standard for GASH's unit and integration tests creation. Our tool is freely available on the GitHub repository[4].

3.2. Architecture

Figure 1 shows the overall GASH's architecture. It is divided into two primary components: **Research Module** and **Analysis Module**. Each component is responsible for specific tasks, enabling the identification of configuration smells in CI/CD pipelines and report generation. Both modules are detailed in the following paragraphs.

The Research Module is designed for research scenarios, leveraging data directly from GitHub repositories. It can receive a single repository URL or a list of repository

[3]https://github.com/prisma/prisma/

[4]https://github.com/mthsfrts/GASH

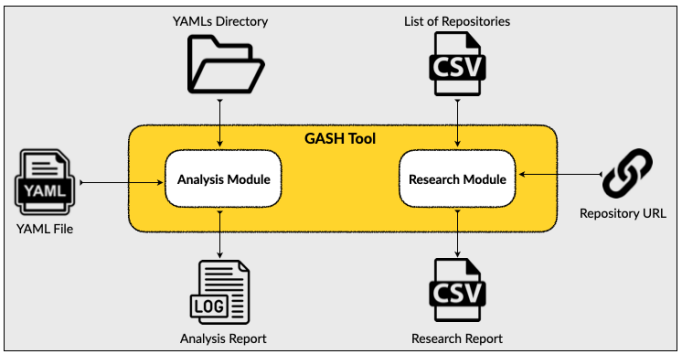

Figure 1. The GASH Tool Architecture.

URLs. This module leverages data mining techniques through the GitHub API and the PyDriller [Spadini et al. 2018] library to extract information from repositories. It analyzes the pipeline's lifecycle and outputs a `.csv` file with the historical data analyzed.

The Analysis Module is designed for development scenarios, allowing developers to verify single or multiple YAML files from one project. The analysis is divided into three levels: workflow, job, and step, allowing for precise and contextualized smell detection. It outputs one or more `.log` files containing the analysis of each YAML file.

3.3. Usage

GASH is configured as a command line tool with a simple, straightforward design. The CLI offers five options (see Table 2), fitting the two use case scenarios described. Each option requires an argument to run correctly. When the `commits` option is used, the user must provide a repository's URL as the next parameter. The option `batch-commits` requires that a `.csv` path be provided containing the list of repositories URLs to be analyzed. The option `repo` is used to select repositories on GitHub based on three criteria provided as parameters: age in years, minimum, and maximum number of stars. GASH searches GitHub to find repositories with a CI/CD pipeline configured using a YAML file and then checks if these repositories match the three criteria. Next, it generates a `.csv` file and behaves likewise the option `batch-commits`. The option `analyze` receives a YAML file path as a parameter while the `batch-analyze` option receives a path pointing to a directory containing all YAML files to be analyzed.

Table 2. Summary of GASH CLI Options

Group	Option	Description
Research	`repo`	Mines repositories based on search criteria, used when a specific repository is not identified.
	`commits`	Analyzes a specific repository's commits for configuration smells.
	`batch-commits`	Studies multiple repositories listed in a CSV file.
Analysis	`analyze`	Performs single-mode analysis on a specific YAML file, providing the file path.
	`batch-analyze`	Analyzes multiple YAML files within a specified directory.

4. Evaluation

4.1. Selected Projects

To evaluate the effectiveness of GASH, a manual analysis was conducted on 15 repositories, yielding a total of 66 YAML files and 446 smells. The chosen repositories match the following criteria: use GitHub Actions, are in activity (last modification in less than one month), from 1 year old to 5 years old, and have the number of stars ranging from 150 to 800. This manual analysis served as a benchmark to compare against the results obtained from GASH, allowing us to compute metrics to evaluate the performance. Table 3 summarizes the selected projects including the number of smells found in each.

Table 3. Summary of Projects used in GASH Evaluation.

Project	#Smells	#Starts	#YAMLs	#Years
Allenact	24	297	3	5
Altswiftui	22	297	3	5
Arena	36	304	5	4
Balerter	16	300	2	5
Bashhub-server	16	303	2	4
Feather	12	302	2	2
Feathub	8	303	5	2
Journalist	14	297	3	3
Neural-speed	22	315	8	1
Nubesgen	62	299	6	4
Prisma	132	38100	15	5
Trigger-workflow-and-wait	14	300	2	5
Typst-physics	18	297	2	2
Upgini	18	309	2	4
Valkyrie	32	303	6	4
Total	446		66	

Although the selected projects have large communities, GitHub Actions is still a relatively new tool. This suggests that older and larger projects are more reluctant to use GitHub's native service as the primary CI/CD tool.

4.2. Performance Metrics

In the evaluation, we employ a set of well-established metrics to assess the GASH performance, which we explain in the following. First, we start with basic metrics: True Positives (TP): correctly identified instances of smells; True Negatives (TN): correctly identified instances where smells are not present; False Positives (FP): incorrectly identified instances of smells; and False Negatives (FN): missed instances of smells that are present. The Precision metric is the proportion of TP identifications made by GASH to the total number of positive identifications (i.e., TP + FP). The Recall metric computes the proportion of TP identifications made by GASH to the total number of actual positive cases in the manual analysis (TP + FN). Finally, the F_1-score is the harmonic mean of Precision and Recall, providing a single metric that balances both concerns. The equations (1), (2), and (3) above precisely describe the derived metrics.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{1}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2}$$

$$\text{F}_1 - \text{score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3}$$

4.3. Results and Discussion

Table 4 summarizes the performance metrics for each type of smell considered in the evaluation. Overall, the results indicate that GASH is highly effective in detecting most predefined smells, with F_1-score values consistently above 0.80 for most smell types. The high precision and recall values demonstrate GASH's reliability in identifying true positives while minimizing false positives and false negatives. The Lack of Error Handling and Unsecure Protocol smells reach the maximum F_1-score (1.00) while the Hard-Coded Secret smell has the lowest F_1-score value (0.65).

Table 4. Performance Metrics of GASH for Detected Smells

Smells Type	Precision	Recall	F1-score	TP	TN	FP	FN
Code Replica	0.81	1.00	0.89	17	35	4	0
Lack of Error Handling	1.00	1.00	1.00	56	0	0	0
Misconfiguration	0.73	1.00	0.85	41	0	15	0
Long Code Blocks	0.62	1.00	0.76	32	4	20	0
Admin By Default	1.00	0.71	0.83	5	49	0	2
Hard-Coded Secret	0.77	0.56	0.65	10	35	3	8
Remote Triggers	0.89	1.00	0.94	25	28	3	0
Unsecure Protocol	1.00	1.00	1.00	4	52	0	0
Untrusted Dependencies	1.00	0.82	0.90	23	28	0	5

Regarding high Recall but lower Precision, the Code Replica, Misconfiguration, and Long Code Blocks detection showed high Recall (1.00) but lower Precision (varying from 0.56 up to 0.82). Thus, the results suggest that GASH successfully identified all actual positives but also made some incorrect positive identifications. In contrast, Untrusted Dependencies smell detection had perfect Precision but slightly lower Recall, indicating no false positives but a few missed actual positives.

The performance metrics for some smells, such as Hard-Coded Secret and Admin By Default, did not achieve perfect scores. This can be attributed to the diverse ways these smells can manifest within CI/CD pipelines, necessitating deeper and more extensive analyses to capture all variations accurately. In contrast, smells that scored 1.00, like Unsecure Protocol, had better detection because their implementation is straightforward — either the parameter is present or not. Although the values set can vary, their inclusion in the code is consistent, facilitating more effective and accurate analysis.

Additionally, varying levels of developer experience and team seniority lead to different implementation styles for pipeline configurations. Less experienced developers may not follow best practices, increasing the incidence of certain smells. Furthermore,

GitHub's account, **free** and **enterprise**, affect pipeline configuration. User accounts operate under free monthly minutes, while enterprise accounts are billed monetarily, encouraging organizations to optimize pipeline configurations to avoid high costs.

4.4. Limitations and Threats to Validity

In this section, we present limitations and threats to the study's validity.

Limitations: The main limitation of this study is that GASH is designed specifically for GitHub Actions and currently detects only nine types of smells. This focus excludes other CI/CD platforms and potential smells. The tool's effectiveness is also limited by the predefined smells and may not capture all nuances in evolving CI/CD practices. Additionally, the manual analysis was conducted on a small subset of repositories, which might not fully represent the diversity in broader software development environments.

Conclusion Validity: Threats to conclusion validity include the potential for incorrect identification of smells by GASH. False positives could skew the analysis and lead to incorrect conclusions about the prevalence and impact of specific smells. However, this threat is mitigated since the tool is more prone to false positives than false negatives. False positives inflate the number of detected smells but are less detrimental than false negative, which could overlook critical issues.

Construct Validity: Threats to construct validity concerns problems with the generalization of the study. There's a possibility of human bias being present when the projects are chosen. To alleviate this, the manual choice of projects was made randomly after the filtering, invalidating only if the project did not fit our criteria. Other parts of our analysis were automated to avoid mistakes and bias.

5. Conclusion and Future Work

In this study, we introduced GASH (GitHub Actions Smell Hunter), a tool designed to identify nine specific types of configuration smells in GitHub Actions CI/CD pipelines, focusing on security and maintenance concerns. Our analysis demonstrated GASH's effectiveness in detecting these smells, thus helping enhance pipeline quality and security.

While GASH effectively identifies straightforward smells like Unsecure Protocol and Lack of Error Handling, it faces challenges with complex smells such as Hard-Coded Secret and Admin By Default. These results underscore the need for continuous refinement and deeper analysis to capture all variations of these smells.

Future work will focus on refining GASH to enhance its detection capabilities further and integrating additional smell categories to improve CI/CD pipeline quality and security. Specific improvement areas that might be included: (i) incorporating additional types of smells, particularly those related to performance and scalability; (ii) utilizing machine learning algorithms and combining static and dynamic analysis; and (iii) conducting large-scale and longitudinal empirical studies.

References

[Kinsman et al. 2021] Kinsman, T., Wessel, M., Gerosa, M. A., and Treude, C. (2021). How do software developers use github actions to automate their workflows? In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 420–431.

[Rahman et al. 2019] Rahman, A., Parnin, C., and Williams, L. (2019). The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 164–175.

[Spadini et al. 2018] Spadini, D., Aniche, M., and Bacchelli, A. (2018). PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 908–911, New York, New York, USA. ACM Press.

[Vasilescu et al. 2015] Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., and Filkov, V. (2015). Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 805–816, New York, NY, USA. Association for Computing Machinery.

[Vassallo et al. 2020] Vassallo, C., Proksch, S., Jancso, A., Gall, H. C., and Di Penta, M. (2020). Configuration smells in continuous delivery pipelines: a linter and a six-month study on gitlab. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 327–337, New York, NY, USA. Association for Computing Machinery.

[Wessel et al. 2023] Wessel, M., Vargovich, J., Gerosa, M. A., and Treude, C. (2023). Github actions: The impact on the pull request process. *Empirical Softw. Engg.*, 28(6).

[Zhang et al. 2024] Zhang, Y., Wu, Y., Chen, T., Wang, T., Liu, H., and Wang, H. (2024). How do developers talk about github actions? evidence from online software development community. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.