GHALogs: Large-Scale Dataset of GitHub Actions Runs

Florent Moriconi
EURECOM
AMADEUS

Thomas Durieux
TU Delft

Jean-Rémy Falleri
University of Bordeaux,
LaBRI, UMR 5800

Raphaël Troncy
EURECOM

Aurélien Francillon
EURECOM

***Abstract*—In recent years, continuous integration and deployment (CI/CD) has become increasingly popular in both the open-source community and industry. Evaluating CI/CD performance is a critical aspect of software development, as it not only helps minimize execution costs but also ensures faster feedback for developers. Despite its importance, there is limited fine-grained knowledge about the performance of CI/CD processes, while this knowledge is essential for identifying bottlenecks and optimization opportunities. Moreover, the availability of large-scale, publicly accessible datasets of CI/CD logs remains scarce. The few datasets that do exist are often outdated and lack comprehensive coverage. To address this gap, we introduce GHALogs, a new dataset comprising 116k CI/CD workflows executed using GitHub Actions (GHA) across 25k public code projects spanning 20 different programming languages. This dataset includes 513k workflow runs encompassing 2.3 million individual steps. For each workflow run, we provide detailed metadata along with complete run logs. To the best of our knowledge, this is the largest dataset of CI/CD runs that includes full log data. The inclusion of these logs enables more in-depth analysis of CI/CD pipelines, offering insights that cannot be gleaned solely from code repositories. We postulate that this dataset will facilitate future CI/CD pipeline behavior research through log-based analysis. Potential applications include performance evaluation (e.g., measuring task execution times) and root cause analysis (e.g., identifying reasons for pipeline failures).**

I. Introduction

Continuous Integration and Continuous Deployment (CI/CD) are software engineering practices encouraged by the DevOps mindset. Continuous Integration promotes the regular integration of new code (e.g., bug fixes, new features) into the main codebase. Automated tests and tools are regularly executed to assess the quality of new code (e.g., linter, static code analysis, compilation). CI aims to reduce the feedback loop, alerting developers quickly when new code does not meet quality requirements. CD encourages regular deployment to the production environment to reduce the number of changes between releases. Hence, it aims to reduce the time required from commit to production and to ease root cause analysis in the event of deployment failure. CI/CD is a widely adopted practice in industry and open-source projects.

Many CI/CD orchestrators, such as GitHub Actions, GitLab, CircleCI, Travis, Drone, and Jenkins, are available. GitHub Actions (GHA) was launched in 2019 and has gained significant interest from open-source projects since many use GitHub to host their source code. GitHub Actions is extensively integrated with GitHub source code management solution, simplifying its adoption and usage. The use of GitHub Actions is free for public projects, significantly contributing to its popularity. However, little is known about the real-world usage of GitHub Actions, primarily because of the lack of large-scale datasets of GHA runs. To address this gap, this paper introduces a large-scale dataset of GitHub Actions runs, including run logs and metadata. The remainder of this paper is organized as follows. In Section II, we present some related work. In Section III, we describe the main components of GitHub Actions. In Section IV, we describe the construction of the dataset and its core properties. Finally, in Section VI, we conclude and outline some future work.

II. Related Work

Numerous studies highlight the widespread adoption and significance of CI/CD. For instance, Hilton et al. [16] noted as early as 2016 that $70\,\%$ of the most popular projects and $40\,\%$ of all GitHub projects employ continuous integration. A study examining tests conducted during CI at Google reveals that, on an average day, $800\,000$ builds and 150 million test runs are executed [18]. The TravisTorrent dataset encompasses logs from millions of Travis CI builds [1], facilitating investigations into the Travis CI platform. More recent Travis datasets, such as those in [3], [11], [12], provide additional metadata with the logs for further analysis. However, Travis is gradually losing popularity among developers, especially for open-source projects. We, instead, focus on GitHub Actions which stands out as the most popular open-source continuous integration service according to [14].

Decan et al. [8] analyze code repositories to study the adoption and practices of GitHub Actions workflows, focusing on workflow automation, action reuse, and related security and versioning aspects. However, their analysis is limited to repository metadata and configuration files; they do not analyze the actual execution of workflows or provide run logs. Cardoen et al. [4] focus on collecting commit histories related to changes in GitHub workflow files. While CI/CD runs capture workflow executions triggered by events, the dataset proposed by the authors tracks changes in the workflow files themselves. As such, it provides insights into how workflows are defined and modified over time. However, it does not provide performance or outcome data of the workflows' executions. Bouzenia et al. [2] propose an empirical study of resource usage and optimization opportunities of GitHub Action workflows based

on run logs. However, they studied a limited number of around 950 repositories while we have collected more than 25k repositories. There is a significant lack of large, publicly available datasets for CI/CD builds that are representative of real-world scenarios. Existing datasets are often limited in scale and diversity, focusing on specific tools or small-scale systems. This limitation hinders broader applicability in optimizing CI/CD processes. Our study aims to tackle this gap.

III. Background

GitHub Actions (GHA) is a full-featured CI/CD orchestrator proposed by GitHub that aims to compete with other major CI/CD systems such as Gitlab, Travis CI, Jenkins, Circle CI, or Drone. GHA is available for both private and public projects on GitHub. GHA is based on runs, workflows, jobs, and steps. A run is an execution of a workflow. A workflow, also referred to as a CI/CD pipeline, is composed of jobs. Each job is started in a fresh environment (i.e., a dedicated virtual machine). Workflows can run jobs in sequence or parallel. Each job contains a sequence of steps (action or shell code). GitHub provides cloud runners (i.e., virtual machines that will execute jobs) based on Linux, Windows, and MacOS. Cloud runners are managed by GitHub and are billed per minute. Free accounts have 2,000 free minutes of workflow execution each month, while paid accounts can choose between different plans to meet their needs. Therefore, there is a strong interest in optimizing CI/CD execution time. Project owners can also set up self-hosted runners for advanced use cases.

A. Workflows

A GHA workflow is a set of jobs that will be triggered on specific conditions. A code repository can have multiple workflows. For instance, a workflow for code quality (e.g., triggered on any code change) and a workflow for releasing (e.g., triggered when a developer adds a new git tag). Workflows are defined using YAML files stored in the code repository (in `.github/workflows`). Workflows define jobs that are started in parallel unless they are configured to depend on other jobs, and implement fail-fast, i.e. when a job fails, all concurrent jobs are stopped to save resources. Indeed, when a job fails, the run is considered failed unless configured otherwise. Jobs consist of steps that are run in order. Output (i.e., stdout, stderr) from steps is saved in the run log. The GitHub runner enriches the run log. For instance, it prints in the log various metadata such as the runner version or the container image that populated the run environment. It also prefixes log lines with a timestamp indicating when the log line was emitted. Run logs are mostly useful when runs are failing: developers often investigate the run log to find the reason for the failure.

B. Steps

Steps can be either Action or shell steps. Unlike jobs, steps are executed in the order they are defined and in the same runner environment. When a step fails, the job is aborted unless configured otherwise. We further explain the shell and Action steps in the remainder of the section.

Shell steps: Shell steps offer great flexibility by allowing developers to use scripting languages that are generally well-known. The default shell interpreter depends on the runner environment (e.g., bash for Linux-based runners, PowerShell for Windows-based runners). Scripting languages allow developers to implement custom CI/CD strategies. For instance, installing dependencies and running tests in the JavaScript ecosystem can be done using the shell command `npm install && npm test`. GHA runners allow the injection into shell code of contextual information using variable interpolation, e.g., git reference that triggered the run, repository secrets, and environment variables. Shell commands return a code enabling GHA to decide if the step has succeeded or failed. In conclusion, step shells are well-known by developers to set up CI/CD pipelines, but they do not offer off-the-shelf features for common CI/CD tasks.

Action steps: GHA encourages the use of actions. Actions are off-the-shelf steps for common CI/CD tasks. This helps to decrease the development effort required to use GitHub Actions. For instance, the `checkout` Action allows the checkout of the repository code in the job environment. By default, only a single commit is fetched for the reference that triggered the workflow. Actions can be customized using a `with` block in the YAML workflow definition file. For the `checkout` action, developers can configure the Action to pull git submodules. Available parameters are usually defined in the Action documentation. Actions can be stored in any GitHub repository, including the repository where the Action will be used. Versioning is done through usual git references (e.g., branch, tag, commit hash). Therefore, the Action version can refer to code that can be modified (e.g., git tag). Unlike shell steps, it might be challenging to ensure the integrity of the code that will be run during workflow execution when using Actions from other repositories. Under the hood, actions can rely on JavaScript code, custom Docker images, or other actions (composite action). However, this is not directly visible to end-users. Therefore, actions are roughly black boxes that perform a specific task. To facilitate the discovery of actions, GHA provides a marketplace. Action owners must register in the marketplace for their actions to be available. As of November 2023, there were 20,782 actions available in the marketplace. However, only 809 actions are coming from verified creators. As actions can access sensitive data such as CI/CD secrets, developers prefer actions from verified creators [19]. Therefore, actions might not be available for less common needs. In this context, developers can fall back to shell steps.

C. Runs and Logs

When workflows are triggered, the jobs are executed by GHA runners. GHA runners collect all the standard output produced as the result of executing the steps in log files. Interwoven with the standard output of steps, GHA runners put dedicated information such as which job and step is running and which environment version has been set up.

```
2023-11-16T22:45:31.2665876Z ##[group]Run actions/checkout@v4.1.1
2023-11-16T22:45:31.2666567Z with:
2023-11-16T22:45:31.2666980Z   repository: home-assistant/core
```

Fig. 1: Extract of GitHub Action log file. Each line is prefixed with the time of execution.

IV. Methodology

Our goal is to provide a dataset that enables large-scale analysis of GitHub Actions ecosystem. We aim to support future studies analyzing runtime properties of CI/CD executions, such as analyzing task execution times, failure patterns, or test results. As a consequence, collecting workflow definition files is not enough, as it is commonly done by the other studies of the GHA ecosystem [5], [9], [17], because they do not contain any runtime information. In this section, we explain our methodology for developing the GHALogs dataset, how we have selected the repositories (Section IV-A) and how we have extracted their run logs (Section IV-B).

A. Selection of repositories

Considering the large number of public repositories in GitHub, we have selected only a subset of repositories relevant to the study. To discover repositories, we leverage GitHub Search [7]. GitHub Search is a search engine for GitHub repositories. It can be used to search for public GitHub repositories using different criteria (e.g., number of stars, license, topics). Using the search engine, we selected repositories with more than 100 stars. This is a common criterion in the literature [15], [20], [6] for selecting repositories with sufficient popularity and visibility. We obtained a list of 239 106 repositories. For each repository with more than 100 stars, we have extracted the number of GHA runs (the sum of runs of all workflows) over the latest 90 days using the GHA API. We successfully scraped 239 106 (99.3 %) repositories. We implemented a retry mechanism to mitigate the impact of transient errors (e.g., HTTP 502 error code). For 0.7 % of the repositories returned by GitHub Search, we could not check for the use of GitHub Actions because of scraping errors: HTTP 404 (0.53 %), HTTP 410 (0.10 %), HTTP 403 (0.06 %), HTTP 502 (0.01 %) and HTTP 451 (>0.01 %). This extraction was done between the 1st and the 5th of October, 2023. We identified that 78 % of repositories have 0 runs, 84 % have 10 or fewer runs, and 98 % have 1000 or fewer runs. We selected repositories with more (greater or equal) than 30 runs in the last 90 days. We identified 28 683 (12 %) repositories. We excluded 361 fork repositories. We aim to exclude toy usage of GHA by selecting repositories with more than one run every three days (on average). It is worth mentioning that some workflows can have less than 30 runs over the 90-day period, e.g., releasing workflows. However, we made the selection at the repository level. If the repository has more than 30 runs over 90 days, we scrape runs of all workflows, even if each workflow has less than 30 runs over the period. Therefore, we identified 28 322 repositories with over 30 runs in the last 90 days. The list of repositories is not updated after the initial construction. In other terms, we did not search for new repositories that could satisfy the run activity criterion after the initial discovery. We identify a *workflow* by its repository and its path in `.github/workflows`. Therefore, if two workflow files in two repositories are identical, we treat them as different workflows.

We scraped all workflows present in the selected repositories. However, we did not scrape all runs for each workflow. Indeed, some workflows have a very large number of runs (e.g., dozens of thousands in 90 days). Therefore, we scraped only the five most recent runs of each workflow. Downloading 5 runs per workflow allows us to compute average values (e.g., step execution time) for the same workflow. This is a balanced approach between getting a highly representative sample of runs and the practicality of data collection. We restricted the download of ZIP archives containing run logs to a maximum size of 20 MB. If the archive is larger than this threshold, the download is aborted, and the run is ignored. We chose a 20 MB size limit as it excludes less than 1 % of runs, thus ignoring only a minimal number of runs. In the following section, we describe the downloading process of runs.

B. Downloading Runs

We aim to discover new runs as fast as possible. Indeed, some of the information we aim to scrape, such as run logs, can be deleted after the run is finished. Run logs are kept for 90 days by default. However, repository owners can reduce the log retention to 1 day. We, therefore, scrape runs as fast as possible after the run is finished. We scaled the number of instances to keep processing time under 1 day. GitHub imposes a rate limit of 5000 API calls per hour per account. As recommended by GitHub [13], we use conditional requests to detect whether a repository contains new runs. This helps to reduce resource usage (i.e., by returning an empty response with HTTP code 304) when no new run is available and avoid rate limiting as conditional requests do not count against rate limits. Conditional requests rely on `Etag` header. We discovered that `Etag` values are related to the token used in the request: different tokens will lead to different `Etag` for the same resource. Consequently, conditional requests should be made using the same GitHub token. The fetcher component loops on repositories to detect if new runs exist compared to runs already scraped, then pushes newly discovered runs for further processing.

For each new run, a worker downloads run metadata and run logs. We only scrape completed runs when the outcome of the job is `"success"`, `"failure"`, or `"timed_out"` in the `"conclusion"` field. Indeed, runs with a different outcome do not have a log, making them irrelevant to this study. GitHub allows users to download run logs as ZIP archives containing logs for jobs and steps. Step logs are redundant with job logs (i.e., step logs are present in job logs). We postulate that GitHub uses this file structure to ease the processing of logs. However, as the ZIP algorithm compresses files independently,

we extract and re-compress log files in a tar archive with gzip compression to reduce the compressed size of the log archive. We kept log archive contents identical to support future usage of the dataset.

C. Analyzing runs

In addition to the run metadata provided by the GitHub API, we parse the job logs. Initially, we extracted structural information from the logs using specific regular expressions. These included details such as the CI container image, GitHub Actions in use, and the shell and GitHub Action steps.

Concerning the shell step, we utilized an existing shell parser [10] designed for analyzing shell scripts within Dockerfiles. This specific parser provides an extended AST, which, in addition to the traditional AST, also provides an abstraction on top of popular command lines. We extended the tool to annotate the commands. For instance, given a shell step: `git clone myrepo;cd myrepo;mvn test;`, it captures the commands `git`, `cd` and `mvn` like any traditional shell parser, and it also identifies command annotations: `GIT-CLONE`, `GIT-REPOSITORY`, `PATH`, `MVN-TEST`. The annotations allow a separation based on subcommands, such as `git clone` and `git diff`, which exhibit different behaviors. The parser supports 91 types of commands[1] with more than 500 annotations.

Our shell step parsing success rate is 96.44 %. Parsing issues occurred due to parser exceptions (3.39 %) or invalid requests (0.27 %). The parser remarkably annotated 70.34 % of all commands. This extensive coverage ensures a comprehensive understanding of the shell steps used in GitHub workflows.

D. Dataset metrics

We identified 28 322 code repositories with more than 30 runs in 90 days, excluding forks. These repositories contain 151k workflows. There are 20 different programming languages. The number of repositories, workflows, and runs analyzed are depicted for each programming language in Table I. GitHub identifies the main programming language used in the repository. We attempted to retrieve 5 runs per workflow. However, we were not able to retrieve exactly 5 runs per workflow. Some workflows are rarely used, thus, there are fewer than 5 runs in the last 90 days. Therefore, run logs have expired and cannot be downloaded. In practice, we retrieved on average 4.41 runs per workflow (std: 1.28). We were able to retrieve at least one run log for 116k workflows (77 %). We collected 513 492 run logs. The total uncompressed size is 640 GB. We focused only on workflows with at least one run log: 116k workflows over 151k workflows. For the following metrics, we consider one run per workflow. Indeed, it maximizes the diversity as runs from the same workflows are likely to share the same jobs and steps. Therefore, from 116 259 runs (i.e., 1 per workflow), we extracted 348 909 jobs. There are, on average, 3.3 jobs per workflow (median: 1.0, std: 10). From jobs, we extracted 2 327 747 steps. There are, on average, 6.7 steps per job (median: 5.0, std: 5.4).

[1]List of supported commands: https://github.com/D2KLab/gha-dataset/blob/master/COMMANDS.md

TABLE I: Breakdown of the programming languages of repositories considered for the study. The number of repositories sorts languages. The number of runs refers to runs where logs were successfully retrieved.

Language	# Repositories		# Workflows		# Runs	
Python	5.52k	19.50 %	22.34k	19.21 %	97.31k	18.95 %
TypeScript	4.44k	15.66 %	18.90k	16.25 %	84.19k	16.40 %
Go	2.95k	10.41 %	15.77k	13.56 %	70.38k	13.71 %
JavaScript	2.78k	9.80 %	9.78k	8.41 %	44.2k	8.61 %
C++	2.14k	7.55 %	9.22k	7.93 %	40.98k	7.98 %
Java	2.07k	7.32 %	8.34k	7.17 %	36.66k	7.14 %
Rust	2.02k	7.14 %	7.97k	6.86 %	34.9k	6.80 %
C	1.25k	4.41 %	5.17k	4.45 %	23.3k	4.54 %
PHP	1.2k	4.22 %	4.56k	3.92 %	20.12k	3.92 %
C#	1.1k	3.88 %	3.79k	3.26 %	15.88k	3.09 %
Shell	735	2.60 %	2.71k	2.33 %	11.95k	2.33 %
Ruby	639	2.26 %	2.3k	1.98 %	10.45k	2.04 %
Kotlin	634	2.24 %	2.27k	1.95 %	9.72k	1.89 %
Dart	319	1.13 %	1.38k	1.18 %	5.93k	1.15 %
Swift	250	0.88 %	250	0.69 %	3.29k	0.64 %
Elixir	126	0.44 %	126	0.30 %	1.56k	0.30 %
Objective-C	60	0.21 %	60	0.24 %	1.21k	0.24 %
Nix	56	0.20 %	56	0.15 %	56	0.15 %
Groovy	36	0.13 %	36	0.12 %	36	0.12 %
Smalltalk	6	0.02 %	6	0.01 %	6	0.01 %
Total	28 322		116 259		513 492	

V. Future Work

Looking forward, there are several promising avenues for future work. Investigating failure patterns will help identify the root causes of CI/CD issues and suggest strategies for improving reliability. Performance analysis, including the study of execution times and optimization opportunities, is critical for identifying bottlenecks and proposing ways to improve the speed and efficiency of workflows. At scale, comparing CI/CD practices across projects can help detect shared usage patterns and dependencies, offering insights into common best practices or configurations that might be applicable across different contexts. Finally, a detailed examination of shell usage within steps and the reasons for not adopting prebuilt GitHub Actions is essential. This could uncover barriers to using actions, such as technical debt, limitations in functionality, or trust concerns, and guide improvements to the GitHub Actions ecosystem.

VI. Conclusion

In this paper, we introduced GHALogs, the largest publicly available dataset of GitHub Actions runs, comprising over 116k workflows from 25k repositories, with 513k workflow runs encompassing more than 2.3 million individual steps. The dataset spans 20 programming languages and includes comprehensive run logs and metadata, offering opportunities for fine-grained analysis of CI/CD processes at workflow, job, and step levels. This establishes a foundation for future research into GitHub Actions and the broader CI/CD landscape.

Dataset Availability

The dataset and the code used in this paper are hosted on Zenodo at https://doi.org/10.5281/zenodo.10154919.

REFERENCES

[1] Moritz Beller, Georgios Gousios, and Andy Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 447–450. IEEE, 2017.

[2] Islem Bouzenia and Michael Pradel. Resource usage and optimization opportunities in workflows of github actions. 2024. 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24).

[3] Carolin E Brandt, Annibale Panichella, Andy Zaidman, and Moritz Beller. Logchunks: A data set for build log analysis. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 583–587, 2020.

[4] Guillaume Cardoen, Tom Mens, and Alexandre Decan. A dataset of github actions workflow histories. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 677–681, 2024.

[5] Tingting Chen, Yang Zhang, Shu Chen, Tao Wang, and Yiwen Wu. Let's supercharge the workflows: An empirical study of github actions. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 01–10, 2021.

[6] Boris Cherry, Pol Benats, Maxime Gobert, Loup Meurice, Csaba Nagy, and Anthony Cleve. Static analysis of database accesses in mongodb applications. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 930–934. IEEE, 2022.

[7] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for MSR studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pages 560–564. IEEE, 2021.

[8] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. On the use of github actions in software development repositories. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 235–245, 2022.

[9] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. On the use of github actions in software development repositories. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022*, pages 235–245. IEEE, 2022.

[10] T. Durieux. Empirical study of the docker smells impact on the image size. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 992–992, Los Alamitos, CA, USA, apr 2024. IEEE Computer Society.

[11] Thomas Durieux, Rui Abreu, Martin Monperrus, Tegawendé F Bissyandé, and Luís Cruz. An analysis of 35+ million jobs of travis ci. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 291–295. IEEE, 2019.

[12] Thomas Durieux, Claire Le Goues, Michael Hilton, and Rui Abreu. Empirical study of restarted and flaky builds on travis ci. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 254–264, 2020.

[13] Github. Best practices for using the rest api. https://docs.github.com/en/rest/guides/best-practices-for-using-the-rest-api.

[14] Mehdi Golzadeh, Alexandre Decan, and Tom Mens. On the rise and fall of ci services in github. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 662–672. IEEE, 2022.

[15] Junxiao Han, Shuiguang Deng, Xin Xia, Dongjing Wang, and Jianwei Yin. Characterization and prediction of popular projects on github. In *2019 IEEE 43rd annual computer software and applications conference (COMPSAC)*, volume 1, pages 21–26. IEEE, 2019.

[16] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 426–437, 2016.

[17] Timothy Kinsman, Mairieli Wessel, Marco A. Gerosa, and Christoph Treude. 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, 2021.

[18] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 233–242. IEEE, 2017.

[19] Sk Golam Saroar and Maleknaz Nayebi. Developers' perception of github actions: A survey analysis. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, page 121–130, New York, NY, USA, 2023. Association for Computing Machinery.

[20] Wei Tang, Zhengzi Xu, Chengwei Liu, Jiahui Wu, Shouguo Yang, Yi Li, Ping Luo, and Yang Liu. Towards understanding third-party library dependency in c/c++ ecosystem. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.