ACTIONSREMAKER: Reproducing GITHUB ACTIONS

Hao-Nan Zhu, Kevin Z. Guan, Robert M. Furth and Cindy Rubio-González
University of California, Davis
United States of America
{hnzhu, zeyguan, rmfurth, crubio}@ucdavis.edu

***Abstract*—Mining Continuous Integration and Continuous Delivery (CI/CD) has enabled new research opportunities for the software engineering (SE) research community. However, it remains a challenge to reproduce CI/CD build processes, which is crucial for several areas of research within SE such as fault localization and repair. In this paper, we present ACTIONSREMAKER, a reproducer for GITHUB ACTIONS builds. We describe the challenges on reproducing GITHUB ACTIONS builds and the design of ACTIONSREMAKER. Evaluation of ACTIONSREMAKER demonstrates its ability to reproduce fail-pass pairs: of 180 pairs from 67 repositories, 130 (72.2%) from 43 repositories are reproducible. We also discuss reasons for unreproducibility. ACTIONSREMAKER is publicly available at https://github.com/bugswarm/actions-remaker, and a demo of the tool can be found at https://youtu.be/flblSqoxeAk.**

***Index Terms*—CI/CD, GitHub Actions, software mining, software build, software reproducibility**

I. INTRODUCTION

Continuous Integration (CI) and Continuous Delivery (CD) play important roles in modern software development. They allow developers to configure and deploy a process to be automatically triggered by designated events (e.g., `git push`). In most cases, the CI/CD process will involve setting up the environment, building the code, and/or running tests. There are various popular CI/CD service providers, including TRAVIS-CI [6], GITHUB ACTIONS [2], and JENKINS [3].

The wide use of CI/CD has opened opportunities for the software engineering (SE) research community. Specifically, it has given researchers access to the build and testing processes of software projects at a large scale. Several research areas within SE, such as fault localization and automated program repair, are evaluated on large sets of successful and failed CI/CD builds. Furthermore, previous work has leveraged CI/CD to mine large scale code repositories and build software defect datasets [12, 15], to predict build outcomes [9, 10], or to automatically repair build scripts [11, 14].

To realize the full value of CI/CD from a research perspective, it is essential to be able to reproduce the build processes at scale whenever needed. Successful reproduction means the ability to run exactly the same steps in exactly the same environments and obtaining exactly the same outcomes as when the process was first completed (e.g., the code was pushed to GITHUB). Unfortunately, the process of reproducing builds is hampered by many challenges such as the unavailability of software dependencies [14, 15]. Previous work [15] proposed a methodology to mine and *reproduce* builds of GITHUB projects that use TRAVIS-CI by using TRAVIS-CI DOCKER images to recreate the exact build environments and TRAVIS-BUILD [5] to automatically create build scripts. While having reproduced thousands of builds, the success rate was extremely low. Most recently, reproducing TRAVIS-CI builds has become even more challenging: following a change in business model, TRAVIS-CI historical logs for a majority of projects are no longer available, and DOCKER images capturing environments are no longer publicly accessible.

As mentioned earlier, other CI/CD services have gained popularity in the past few years. Among the existing CI/CD service providers, GITHUB ACTIONS [2] is one of the most popular due to its deep integration with GITHUB, which is the most popular software project hosting provider with more than 350 million code repositories [1]. GITHUB ACTIONS uses *workflow runs* to refer to build processes. However, to reproduce workflow runs for GITHUB ACTIONS, there are several challenges. First, GITHUB ACTIONS by default uses Azure VMs to host workflow runs so it is not easy to reconstruct the exact runtime environment while ensuring portability. Second, unlike TRAVIS-CI, GITHUB ACTIONS does not provide an official converter from workflow files to build scripts, thus a parsing process is required to run a workflow "offline." To address these challenges, it requires (1) proper containerization techniques to reconstruct the runtime environment of a build, and (2) a sophisticated parser to convert the non-executable workflow file to executable build scripts.

In this paper, we demonstrate ACTIONSREMAKER, a reproducer for GITHUB ACTIONS workflows. Given a desired workflow job to reproduce, ACTIONSREMAKER will first reconstruct the original runtime environment by building DOCKER containers with the information retrieved from the original workflow job. Then GITHUBBUILDER translates the workflow file into a build script. Finally, ACTIONSREMAKER packages the software repository and build script along with the required environment into a DOCKER image. To the best of our knowledge, ACTIONSREMAKER is the first tool designed to reproduce existing GITHUB ACTIONS workflow runs.

We evaluate ACTIONSREMAKER by examining its capability to reproduce fail-pass build pairs. Fail-pass pairs are consecutive builds where the first fails and the second passes, and have been used to construct software defect datasets such as BUGSWARM [15]. BUGSWARM is a software defect dataset mined from TRAVIS-CI builds, of which workflows are defined as `.yml` files that can be translated to shell scripts by the official converter provided by TRAVIS-CI. We reuse the pipeline from BUGSWARM to *mine* fail-pass pairs from

979-8-3503-2263-7/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE-Companion58688.2023.00015

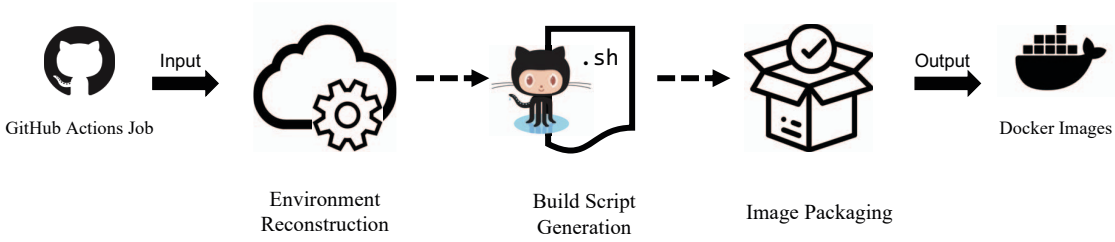

Fig. 1. Steps of ACTIONSREMAKER

GITHUB projects. Among 180 fail-pass pairs mined from 67 projects, ACTIONSREMAKER successfully reproduced 130 (72.2%) of them, which is dramatically higher than the reproducibility reported by BUGSWARM (5.56%) when reproducing TRAVIS-CI builds. We also study and categorize the root causes of unreproducible pairs. The rest of the paper presents background on GITHUB ACTIONS, the approach of ACTIONSREMAKER, and its evaluation.

II. BACKGROUND: GITHUB ACTIONS

GITHUB ACTIONS is a CI/CD service provided by GITHUB. All configurations and steps are defined in `.yml` formatted files in the `.github/workflows` directory of the repository, which are referred as workflow files. Each of workflows will contain events, jobs, steps, and runners configured by developers. Events are specific activities that trigger a workflow run, such as a *push* to a branch or a *pull request*. In a workflow run, there could be multiple jobs for different environments. Each job consists of a set of steps to be executed. Steps are the smallest portable unit in GITHUB ACTIONS. They can be either *predefined actions* or *custom actions* from GITHUB or developers. Runners are the virtual machines where jobs are executed. Developers can use matrix parameters to define a set of environments by combinations of different operating systems, programming languages, and compiler versions for runners. A workflow run triggered by an event executes all jobs defined in the workflow file. Finally, each job executes all steps defined in the job in a runner. The whole process is referred as a *build*.

III. APPROACH

In this section, we describe ACTIONSREMAKER's approach to reproduce GITHUB ACTIONS builds. As shown in Figure 1, ACTIONSREMAKER takes as input the desired GITHUB ACTIONS job, and outputs a DOCKER image in which the job can be easily reproduced. ACTIONSREMAKER first reconstructs the environment based on information retrieved from the original job. Then, GITHUBBUILDER generates a build script based on the workflow file. Finally, with the original environment, ACTIONSREMAKER packages the build script and software repository into a DOCKER image. We discuss limitations of ACTIONSREMAKER at the end of the section.

A. Environment Reconstruction

Given the job ID, ACTIONSREMAKER can retrieve the environmental information via the GITHUB API to reconstruct the original runtime environment. The reconstructed environment includes the operating system, software repository, predefined actions, and the original logs of the job.

For GITHUB ACTIONS, operating systems are specified in the workflow file with the `runs-on` option. Based on this value, ACTIONSREMAKER will determine the original operating system to construct with. However, if the users specify `ubuntu-latest` as the operating system version, ACTIONSREMAKER will parse the original log to get the correct operating system version because `ubuntu-latest` could vary based on the time when the original job was triggered. Next, ACTIONSREMAKER downloads the software repository with the exact revision that corresponds to the job. The GITHUB ACTIONS official runner requires developers to use the `actions/checkout` action at runtime to set up the repository. For ACTIONSREMAKER, it retrieves the project source code by cloning the project and runs `git reset` to reset the project back to the target commit. Sometimes the target commit is not in the git history thus it is not resettable. In such cases ACTIONSREMAKER downloads the zip archive of the commit directly from GITHUB. ACTIONSREMAKER also retrieves the source code for all predefined actions based on the action name and version listed in the job's workflow file. Moreover, since the goal of ACTIONSREMAKER is to reproduce existing workflow runs, it also downloads the original build logs from GITHUB API for future validation.

B. Build Script Generation

GITHUBBUILDER, a core component of ACTIONSREMAKER, automatically generates shell scripts from the workflow file for running a GITHUB ACTIONS job. GITHUBBUILDER retrieves additional workflow metadata like `actor` (the user that triggered the workflow run) and `head ref` (the source branch of the pull request) via the GITHUB API. It also retrieves runtime information like global environment variables, default shell, and working directory from the workflow file. If a job was triggered by a pull request (PR), the PR number is crucial for job reproduction since oftentimes the job will require information such as the title and head repository name of the PR. However, the PR number is not accessible from the API. To address this, ACTIONSREMAKER

parses the original build log to extract the PR number, then uses the GITHUB API to fetch all PR-related information.

GITHUB ACTIONS workflow files contain a list of steps for a job. A step could be either a custom action (specified using the "run" field) or a predefined action (specified using the "uses" field). Regardless of action type, GITHUBBUILDER will extract metadata such as the step name, step number, the conditions to run the step, etc. For custom actions, GITHUBBUILDER will save the user-defined commands into a script file, then add a shell command that executes this script into a main build script. For predefined actions, GITHUBBUILDER supports JavaScript and composite actions. To reproduce those, GITHUBBUILDER first downloads the source code of the predefined action and its metadata file based on the version listed in the workflow file. Then, to reproduce JavaScript predefined actions, it will use Node.js to run the main program specified in the action metadata file. Just like GITHUB ACTIONS, GITHUBBUILDER also passes arguments to JavaScript programs using a list of environment variable. For example, the `actions/setup-java` action requires users to specify the `java-version` option. So if we have `java-version: 17`, it will set the `INPUT_JAVA-VERSION` variable to 17. A composite action is similar to the workflow file; it contains a list of custom and predefined action steps, which can be grouped. To implement the composite action, ACTIONSREMAKER recursively generate the build scripts for this list of steps, similar to how it generates build commands from the steps list in the workflow file.

When generating build scripts, GITHUBBUILDER needs to handle *contexts* and *expressions* used in workflow files. Contexts are the collection of variables containing information about the workflow run, steps, environment variables, runner, etc., and expressions are the combinations of literal values, contexts, operators, and functions. For example, the `startsWith(searchString, searchValue)` function will return `true` if the searchString starts with searchValue. They are generally used in the conditional statements to control step execution. GITHUBBUILDER will replace static contexts (constant variables such as `github.repository`) with strings and replace dynamic contexts (non-constant variables such as `steps.<step_id>.outcome`) with environment variables. GITHUBBUILDER then converts expressions into lists of arguments to evaluate them at runtime.

C. Image Packaging

The final step is to build the DOCKER image. In this step, ACTIONSREMAKER generates a DOCKER file to specify a list of instructions for the image-building process. Some instructions include: adding a new user with proper permission, changing file permissions, adding the source code of the repository and pre-defined actions, and adding the build scripts. We also set the base image of the DOCKER file based on the job's operating system. Currently, ACTIONSREMAKER supports base images `ubuntu-18.04`, `ubuntu-20.04`, and `ubuntu-22.04`.

D. Limitations of ACTIONSREMAKER

ACTIONSREMAKER does not support certain types of actions: (1) actions that invoke DOCKER are not supported since the runtime environment is constructed within a DOCKER container and running DOCKER in DOCKER will lead to security risks, (2) actions that require secret context are not supported since secrets are private and not accessible to the public, (3) `actions/checkout` actions with specified submodules is not supported only when the job is not resettable, since the downloaded zip archive does not contain the `.git` directory.

IV. EVALUATION

In this section, we evaluate ACTIONSREMAKER by measuring its capability to reproduce fail-pass pairs from open-source projects, i.e., we include real-world failures. We answer the following research questions:

RQ1 How effective is ACTIONSREMAKER at reproducing real-world builds?

RQ2 What are the root causes of unreproducible builds?

A. Experimental Setup

We reuse BUGSWARM's pipeline for *mining* fail-pass pairs, which we extended to support GITHUB ACTIONS. In the case of GITHUB ACTIONS, fail-pass pairs are two consecutive job runs where (1) the first job comes from a failed build and the second comes from a passed build, (2) the two jobs are generated by the same GITHUB ACTIONS workflow file and have identical configurations and matrix parameters, and (3) the commits corresponding to each job are from the same branch. To verify reproducibility, we slightly modify BUGSWARM's log analyzer to make it compatible with GITHUB ACTIONS log formats, so it can analyze and compare the reproduced log against the original log.

B. RQ1: Effectiveness at Reproducing Real-World Builds

We use two sets of real-world fail-pass pairs to create our benchmark. The first set consists of manually-selected fail-pass pairs that make use of important and popular features of GITHUB ACTIONS. For instance, we chose pairs from the repository `Netflix/spectator` because its jobs dynamically modify environment variables when they run, and we chose pairs from `junit-team/junit5` because they use composite actions. Such features are expected to be handled by ACTIONSREMAKER. The manual selection yielded 87 fail-pass pairs from 11 repositories.

The second set is generated by mining GITHUB repositories with the adapted BUGSWARM pipeline. For a repository to be considered, it should (1) use GITHUB ACTIONS, (2) be written in Java, and (3) have at least 50 stars on GITHUB.[1] The above criteria yielded a set of over 3,600 repositories. To keep the benchmark at a practical size, we selected 75 of these repositories to be included in the benchmark. Then, we filtered out the pairs with inaccessible original logs, pairs with jobs

[1] ACTIONSREMAKER is language agnostic, but we focused on Java because it is a popular language, and is supported by the BUGSWARM analyzer.

TABLE I
FAIL-PASS PAIR COUNTS FOR EACH TYPE OF UNREPRODUCIBLE PAIR

Category #	Failure Reason	Number	Percentage
1	Test Mismatch	23	12.8%
2	Missing Requirement	10	5.6%
3	Dependency Error	8	4.4%
4	Tool Failure	6	3.3%
5	Other	3	1.7%
	Total	50	27.8%

that run on unsupported operating systems (Section III-C), and pairs with unsupported actions (Section III-D). This excluded 19 out of 75 repositories that have no suitable pairs to mine. Since we wanted to collect a diverse set of pairs while keeping the size of the benchmark manageable, we only kept the three most recent pairs from repositories that had more than 3 suitable pairs, and all pairs from repositories with more than one but less than 3 pairs. The process above generated a set of 93 fail-pass pairs from 56 repositories. Combining the two sets yields a benchmark of 180 fail-pass pairs from 67 repositories.

We provide each fail-pass pair in the benchmark as input to ACTIONSREMAKER, get the resulting DOCKER image, run the generated build script in the DOCKER container, and collect build outcomes. Then we use the adapted BUGSWARM analyzer to extract the number of tests run, passed, failed, and skipped, as well as the name of failed tests. We repeat the process for the original logs and compare them against the reproduced outcomes. A fail-pass pair is reproducible only if the outcomes from both jobs match their original counterparts. Out of the 180 fail-pass pairs in the benchmark, 130 (72.2%) are reproducible. Out of the 67 repositories represented in our benchmark, there were 39 with all pairs reproducible, and 43 with at least one pair reproducible.

C. RQ2: Root Causes of Unreproducible Builds

In total, 50 (27.8%) out of the 180 fail-pass pairs were not reproducible. Our manual inspection identified four reasons for unreproducibility, which are summarized in Table I.

a) Test Mismatches: 23 pairs failed to reproduce because their number of failed and passed tests did not match those listed in the original logs. The main cause for this is test flakiness. Flaky tests are tests that fail in some runs but pass in others. For example, the test `testSingleValueRandomJoin` from the Apache Lucene project failed in the original GITHUB ACTIONS workflow run but ACTIONSREMAKER does not always trigger the expected failure. While the source of flakiness can usually be attributed to non-determinism, it is often difficult to ascertain the exact root cause.

b) Missing Requirements: 10 pairs failed to reproduce due to missing system requirements such as MySQL and the Android SDK. For purposes of practicality, our base images only contain a subset of the software installed on GITHUB ACTIONS's official virtual machines, thus jobs using not included software will not be reproduced correctly.

c) Dependency Errors: Some jobs rely on external services or data that are outside of our control. A total of 8 pairs failed to reproduce due to the inability to access Maven dependencies that are no longer available and external APIs.

d) Tool Failures: Sometimes ACTIONSREMAKER fails to package a job properly; for instance a `git reset` might fail unexpectedly. A total of 6 pairs failed due to this reason.

There were also 3 unreproducible fail-pass pairs that did not fit neatly into any one of above categories. The unreproducible jobs in those pairs failed due to, respectively, the job not being run in a clean Git repository, a Maven plugin failing for unclear reasons, and a syntax error in a predefined action.

V. RELATED WORK

Mining CI Services for Dataset Creation. Among popular CI services, TRAVIS-CI in particular has been mined extensively by the SE community for different purposes. Related to dataset creation, TRAVISTORRENT [7] mines TRAVIS-CI to create a large-scale dataset of TRAVIS-CI builds that allows to easily access information from the build logs as well as trigger commits and aggregated CI project data. BUGSWARM [15] and BEARS [12] mine TRAVIS-CI to automatically create software defect datasets. To the best of our knowledge, ACTIONSREMAKER is the first tool that can be integrated into a pipeline that mines GITHUB ACTIONS to automate the process of *reproducing* builds.

Reproducing and Fixing Builds. Prior work has studied the reproducibility of builds on DOCKERFILE [8], snapshots of Java projects [13], or Python projects [14]. In the scenario of CI services, BUGSWARM [15] reproduced TRAVIS-CI builds with a reported 5.56% success rate. Others have also proposed approaches to fix broken builds [9, 11, 16]. Unlike the above, this is the first study on the reproducibility of GITHUB ACTIONS builds.

***Triggering* GITHUB ACTIONS.** Open source work [4] developed by the community provides functionality to trigger GITHUB ACTIONS workflow runs locally. However, the approach is designed to run a job that has *not* been triggered yet, not for reproducing *historical* workflow jobs. Unlike the above, ACTIONSREMAKER is the first tool targeting the reproduction of existing workflow jobs, and delivering a *reproducible* DOCKER image for future use.

VI. CONCLUSION

We present ACTIONSREMAKER, a tool for reproducing GITHUB ACTIONS workflow runs. ACTIONSREMAKER takes as input the desired workflow job to reproduce and outputs a DOCKER image with the exact environment, software repository and build script. Our evaluation showed that ACTIONSREMAKER can reproduce 130 (72.2%) fail-pass build pairs from 43 out of 67 distinct open-source repositories.

ACKNOWLEDGMENT

This work was supported by the National Science Foundation under award CNS-2016735.

REFERENCES

[1] GitHub Search for Number of Repositories. https://github.com/search?type=repositories, 2022.

[2] GitHub Actions. https://docs.github.com/en/actions, 2022.

[3] Jenkins. https://www.jenkins.io/, 2022.

[4] nektos/act. https://github.com/nektos/act, 2022.

[5] Travis Build. https://github.com/travis-ci/travis-build, 2022.

[6] Travis CI. https://www.travis-ci.com/, 2022.

[7] M. Beller, G. Gousios, and A. Zaidman. Travistorrent: synthesizing travis CI and github for full-stack research on continuous integration. In *MSR*, pages 447–450. IEEE Computer Society, 2017.

[8] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall. An empirical analysis of the docker container ecosystem on github. In *MSR*, pages 323–333. IEEE Computer Society, 2017.

[9] F. Hassan. Tackling build failures in continuous integration. In *ASE*, pages 1242–1245. IEEE, 2019.

[10] F. Hassan and X. Wang. Change-aware build prediction model for stall avoidance in continuous integration. In *ESEM*, pages 157–162. IEEE Computer Society, 2017.

[11] F. Hassan and X. Wang. Hirebuild: an automatic approach to history-driven repair of build scripts. In *ICSE*, pages 1078–1089. ACM, 2018.

[12] F. Madeiral, S. Urli, M. de Almeida Maia, and M. Monperrus. BEARS: an extensible java bug benchmark for automatic program repair studies. In *SANER*, pages 468–478. IEEE, 2019.

[13] M. Maes-Bermejo, M. Gallego, F. Gortázar, G. Robles, and J. M. González-Barahona. Revisiting the building of past snapshots - a replication and reproduction study. *Empir. Softw. Eng.*, 27(3):65, 2022.

[14] S. Mukherjee, A. Almanza, and C. Rubio-González. Fixing dependency errors for python build reproducibility. In *ISSTA*, pages 439–451. ACM, 2021.

[15] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González. Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes. In *ICSE*, pages 339–349. IEEE / ACM, 2019.

[16] H.-N. Zhu and C. Rubio-González. On the reproducibility of software defect datasets. In *ICSE*. IEEE, 2023.