Data Descriptor

Large-Scale Dataset of Local Java Software Build Results

Matúš Sulír *, Michaela Bačíková, Matej Madeja, Sergej Chodarev and Ján Juhár

Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, Slovakia; michaela.bacikova@tuke.sk (M.B.); matej.madeja@tuke.sk (M.M.); sergej.chodarev@tuke.sk (S.C.); jan.juhar@tuke.sk (J.J.)

* Correspondence: matus.sulir@tuke.sk

Received: 12 August 2020; Accepted: 19 September 2020; Published: 21 September 2020

Abstract: When a person decides to inspect or modify a third-party software project, the first necessary step is its successful compilation from source code using a build system. However, such attempts often end in failure. In this data descriptor paper, we provide a dataset of build results of open source Java software systems. We tried to automatically build a large number of Java projects from GitHub using their Maven, Gradle, and Ant build scripts in a Docker container simulating a standard programmer's environment. The dataset consists of the output of two executions: 7264 build logs from a study executed in 2016 and 7233 logs from the 2020 execution. In addition to the logs, we collected exit codes, file counts, and various project metadata. The proportion of failed builds in our dataset is 38% in the 2016 execution and 59% in the 2020 execution. The published data can be helpful for multiple purposes, such as correlation analysis of factors affecting build success, build failure prediction, and research in the area of build breakage repair.

Dataset: http://doi.org/10.17605/OSF.IO/UMK3W

Dataset License: CC-BY 4.0

Keywords: build tool; program compilation; failure; Ant; Maven; Gradle

1. Introduction

There are many possible situations when a person would like to build a third-party software system from source code. For instance, an industrial developer can try to fix a bug in a library used by the system he or she is developing. A student may want to contribute to a popular open source application. A researcher could be searching for applications suitable for experiments. In all of these situations, the person downloads the source code from a software forge such as GitHub and tries to build it using the supplied build script. However, this process often ends with a failure.

Numerous studies focus on build system problems. Neitsch et al. [1] explored build system issues in a few selected multi-language software projects. Both Kerzazi et al. [2] and Seo et al. [3] studied build errors in one specific software company. Tufano et al. [4] studied multiple snapshots of 100 projects, but the study was limited to one build system (Maven). Rabbani et al. [5] analyzed developers' builds from Visual Studio; however, the studied dataset contained only integrated development environment (IDE) events of a limited number of projects without full build logs. Horton and Parnin [6] evaluated the executability of short code snippets called Gists.

Multiple researchers studied continuous integration (CI) builds. Rausch et al. [7] analyzed Travis CI build logs of 14 open source projects. Zolfagharinia [8] studied CI build failures of Perl software

systems. Ghaleb et al. [9] focused on the long duration of CI builds. Another study focused specifically on compiler errors [10].

There exist also datasets related to build systems. TravisTorrent [11] is a collection of a large number of Travis CI build logs and associated metadata for open source projects from GitHub. However, CI builds differ from local builds (on developers' computers) since they are configured with a special file describing the environment and necessary build commands for each project[1]. Furthermore, in the mentioned study, the prevalence of the Travis CI usage among the inspected projects was only 28% [11], limiting the scope of analyzable systems (in our Java-only dataset, the proportion is even lower—18%). The Continuous Defect Prediction (CDP) dataset [12] is an extension of TravisTorrent including file-level change details and software metrics. LogChunks [13] contains 797 Travis CI build logs manually annotated with labels denoting important parts.

However, none of the mentioned studies and datasets includes local (non-CI) build results and logs of a large number of open source projects. Therefore, in this paper, we describe a dataset resulting from a study trying to build thousands of Java projects from GitHub. For each project, our script (i) downloaded the source code, (ii) determined, which build tool (Gradle, Maven, or Ant) does the project use, and (iii) tried to execute the corresponding command. The console output, exit status, and various metadata were recorded.

In our workshop paper from 2016 [14], we performed such a study for the first time. We calculated the proportion of failing builds, determined the most common failure types, and analyzed the correlation between the success/failure and the project's properties. The current data descriptor paper significantly complements our previous study in multiple aspects. First, we have executed the study once more in 2020 on a newly sampled set of projects and using newer software versions. Second, we are publishing the complete dataset for both execution years, including log files which were previously not publicly available. Third, in this paper we provide a description of the data formats for files produced by our study. Finally, we provide a brief comparison of the results from the 2016 and 2020 execution.

2. Method

In this section, we will describe the method used to produce the data files. We aimed to use the same method both in the 2016 and 2020 execution, with a few exceptions, which will be noted when necessary. The complete source code of the scripts that were executed to produce the data files is available at https://github.com/sulir/build-study.

The main part of the study (Sections 2.1–2.5) can be easily reproduced thanks to Docker[2] by running the following command:

```
docker run -itv /results/dir:/root/build quay.io/sulir/builds 10000
```

The final analysis step (Section 2.6) can be performed by running the supplied shell script (`results.sh` in the repository).

2.1. Inclusion Criteria

We were interested in projects from GitHub (excluding forks) matching the following criteria:

- C1: The project is written in Java. This language was selected because of its popularity and portability. Furthermore, it is a compiled language with multiple mature and widely used build

[1] Since CI services are designed to be run on a server, a person trying to build a third-party application usually does not look into CI configuration in order to mimic the CI environment locally. Even if yes, such mimicking can be imprecise. While Docker aims to solve this problem by offering uniform environments, its current prevalence in the projects in our dataset is less than 3%.

[2] https://www.docker.com.

systems. The main programming language of projects was determined using the GitHub API[3] (application programming interface).

- C2: The repository contains a recognizable open source license. By definition, we were interested only in open source projects, not in personal backups, proprietary software, etc. The project's license was determined using GitHub API[4].
- C3: The repository was forked at least once. Thanks to this criterion, we selected only projects whose potential for a cooperation effort has already been shown—some person probably already had the intention to build the project from source. Furthermore, it lowered the number of searched projects to make the selection of random projects technically feasible (since the GitHub API does not support random project selection, we had to download the metadata of all matching projects).
- C4: The project does not use the following technologies: Java Native Interface (JNI), Java Micro Edition (ME), or Android. To maximize internal validity, we aimed to focus on pure Java and excluded projects utilizing separate ecosystems. The mentioned technologies also require manual installation of third-party packages (sometimes with proprietary licenses), which we tried to avoid. The use of the mentioned APIs was determined by searching for files having the particular extensions and containing the text specific for the technology, as described in Table 1.

Table 1. The list of excluded technologies.

Technology	File Name Pattern	File Content
Java Native Interface	*. {c,cc,cpp,cxx}	JNIEXPORT
Java Micro Edition	*.java	import javax.microedition.
Android	*.java	import android.
	AndroidManifest.xml	—

We did not limit the creation or update date of the projects. This means the 2016 execution contains projects updated in years 2008 (the launch of GitHub) to 2016; the 2020 execution contains projects updated in 2008–2020.

2.2. Project Downloading

First, our script downloaded the metadata for all repositories matching C1–C3 using the GitHub Search API[5]. Next, tarballs (.tar.gz archives) of the default branch (usually `master`) were downloaded for random projects from the set; they were extracted, and C4 was tested for them until 10,000 matching projects were found.

This way, we obtained 10,000 random projects matching the criteria in the 2016 execution and another 10,000 ones in the 2020 execution. The samples were independent, which means there is only a small random overlap—less than 8% of the projects from one execution are present also in the second one (some of the projects were updated between the executions, though).

2.3. Build Tool Selection

The next task was to determine which build system was used by a particular project. There are three common build systems used by Java projects: Gradle, Maven, and Ant. Each of them uses a particularly named configuration file: `build.gradle` (Gradle), `pom.xml` (Maven), `build.xml` (Ant). Our script searched for these files in the root directory and assigned the corresponding build system to the project.

---

[3] https://docs.github.com/en/rest/reference/repos.
[4] https://docs.github.com/en/rest/reference/licenses.
[5] https://docs.github.com/en/rest/reference/search.

If a project contained multiple different configuration files, Gradle had a higher precedence than Maven; Ant had the lowest priority. This logic is based on age: We assume a newer build system replaces the older one, while the old configuration file remains there for compatibility or as a partial configuration file for some tasks.

2.4. Environment

The Docker container used to perform the study aimed to simulate a lightweight development environment of a Java programmer. It consisted of a Linux distribution, Java Development Environment (JDK), the three studied build systems (Gradle, Maven, and Ant with its dependency manager Ivy), and Git. See Table 2 for the list of specific versions. Our aim was to include the newest stable version of each software package at the time of the given study execution (the newest long-term support version in the case of JDK). The Docker container also included auxiliary utilities: `tar`, `unzip`, and a Ruby interpreter (for the control script).

Table 2. The virtual environment used to build the projects in the study.

Software	Version (2016 Run)	Version (2020 Run)
Fedora	23	32
OpenJDK	8	11
Gradle	2.14	6.5.1
Maven	3.3.9	3.6.3
Ant	1.9.7	1.10.8
Ivy	2.4.0	2.5.0
Git	2.5	2.26

Since we use a fixed set of software versions, it is probable that a portion of the builds will fail because of a mismatch between the version the project expected and our installed software version. Note, however, that the main purpose of this study is to simulate this situation: A person has certain software installed on the computer (probably the newest versions since software tends to auto-update these days). This person finds a specific software project on GitHub and needs to build it—without contemplating whether this project was updated many years ago or which specific versions of software it requires. The 2016 execution of our study represents such a person in 2016, the 2020 run represents the current situation.

2.5. Build Tool Execution

For each project, a build command was executed based on its detected build tool. According to Spolsky [15], a software system should be buildable in one step, i.e., by executing an appropriate command[6] using the build tool. Based on this premise, we constructed a list of commands corresponding to the individual build tools, visible in Table 3. Our goal was to build a complete software package, typically a JAR file. Test execution was disabled (when this option was available) since it is out of the scope of this study—it is often possible to inspect, modify and run the software even if some of the tests are failing.

The standard output and error streams were redirected to a log file. To prevent a potentially infinite execution, the build time of each project was limited to one hour. The exit code of the process was recorded, a zero value meaning success, 124 meaning timeout, and any other non-zero value meaning failure.

After the build process of each project was finished, the build result along with the corresponding metadata was written to a CSV (comma-separated values) file `results.csv`.

[6] Gradle builds can be executed by Gradle Wrapper—the `gradlew` script in the given repository. However, since a very large portion of these scripts was incompatible with Java 11, we decided to use the installed Gradle version instead.

Table 3. Build tools and the corresponding commands.

Build tool	Command
Gradle	`gradle clean assemble --no-daemon --stacktrace --console=plain`
Maven	`mvn clean package -DskipTests --batch-mode`
Ant	`ant clean; ant jar || ant war || ant dist || ant`

2.6. Error Analysis

After collecting the raw log files and exit codes, we performed a brief analysis of log files of failing projects. First, we automatically assigned a build-tool-dependent *error type* to each log.

In the case of Maven, the error logs contain a URL (Uniform Resource Locator) explaining the main error that occurred. If the log contained multiple URLs, we selected the last one since it tends to be the most specific. Such a URL encompasses also the exception name, from which we removed the "Exception" suffix. This represents the error type, e.g., "DepencencyResolution" or "UnresolvableModel". Three exception classes were too general: MojoFailure, MojoExecution, and PluginExecution. To make them more specific, we appended the failed goal name to them, e.g., "MojoFailure:maven-compiler-plugin".

For Gradle, we captured the thrown exceptions too. If the particular exception was chained (A caused by B caused by C...), we selected the first most direct cause of the root exception (B in this example). The "Exception" suffix was then stripped, thus forming the error type string. Finally, for vague exceptions (Gradle, TaskExecution), the task name was appended ("Gradle:javadoc"). For the "PluginApplication" error type, we appended the plugin name ("PluginApplication:ShadowJavaPlugin").

From Ant logs, we extracted the last executed target name—i.e., the name of the failing target. We prepended the text "Target:" before it, for instance, "Target:compile". If the build failed before even one target was executed, the error type was determined by searching for specific patterns in the log file. For example, "taskdef class ... cannot be found" was mapped to the error type "TaskdefNotFound".

Combined from both the 2016 and 2020 execution, there were 432 unique error types. We selected some of the more frequent types and manually mapped them to human-readable, tool-independent *error categories*, such as "Java compilation" or "documentation generation". Our goal was not to investigate the root causes of failures, but rather to divide the error symptoms into meaningful categories to obtain an overview of the data. The mapping was performed by inspecting examples of error logs and consulting online documentation. First, one author labeled a subset of error types and logs, creating a list of possible categories. Then the rest of the authors categorized portions of error types individually. Doubts were resolved by discussions with other authors. After the labeling was finished, the main author walked through all labels to ensure consistency. If an error type was used very infrequently or when no consensus could be achieved, we left it "uncategorized".

For a build whose error category was "Java compilation", we also automatically extracted the first line of the Java compiler error message, such as "unmappable character (0xC3) for encoding US-ASCII", from the log file.

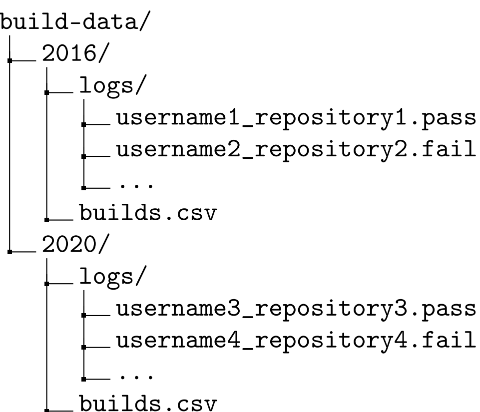
The result of the error analysis is written to the file `builds.csv` which contains the same data as the original `results.csv` file, but also the error types and categories for individual projects.

3. Data Format Description

The direct link to download the data file is https://osf.io/w6fnr/download. In this section, we will describe its contents.

3.1. Directory Structure

The data file is a standard tar-gzipped archive. After extraction, the total size is about 440 MB and the content has the following directory structure:



The top-level directory is divided into directories labeled by the two years of the study execution. Each of these two directories then has a similar structure. The file `builds.csv` is the main results file—it contains project names in the form "username/repository", exit codes, and various metadata. The logs are contained in the `logs` directory, named in the form "username_repository.pass" for successful builds and "username_repository.fail" for failed and timed-out builds.

3.2. Results File

The `build.csv` files are standard comma-separated (CSV) files. The first line represents the header, i.e., column names. The fields are separated by commas, quotes are not present unless necessary. Boolean values are represented in the notation recognized by the R statistical language: T means true, F false. In Table 4, there is a list of all fields along with their description. For a list of possible error categories, see Table 5.

Table 4. Fields of the build.csv file.

Field Name(s)	Description
name	the project name (user/repo)
status	the build success (T) or failure (F); NA for timeouts and projects without a recognized build system
code	the exit code of the build process: 0 (success), 124 (timeout), other non-zero (failure), or NA (no recognized build system)
tool	the main recognized build system used to build the project (Ant, Gradle, Maven, NA)
stars, forks	the number of stars/forks of the repository on GitHub
created	the creation date of the repository (e.g., 2017-09-24 11:17:28 UTC)
pushed	the date of the last source code modification (Git push)
Ant, Buildr, Gradle, Make, Maven, SBT	the presence of the given build configuration files in any directory: T/F
Ivy	the presence of a configuration file of the Ivy dependency manager: T/F
Eclipse, IntelliJ IDEA, NetBeans	the presence of IDE (integrated development environment) project files: T/F
Travis CI	the presence of Travis CI configuration file, `.travis.yml`, in the root directory: T/F
Git submodules	the use of Git submodules (sub-projects in directories): T/F
in_files	the total number of files in the repository before the build process started
out_files	the total number of files after the build finished
new_files	out_files − in_files
error_category	a human-readable, tool-independent build error category from Table 5; may be "uncategorized"
error_type	an automatically derived error type specific for a given build tool; e.g., "Target:-do-compile"
compiler_message	an automatically extracted compiler message—e.g., "package org.junit does not exist"; NA if the error category is not "Java compilation"

Table 5. A list of possible error categories.

Category	Description
build files parsing/interpretation	XML parsing or build script interpreter error, including plugins (e.g., a missing method)
configuration	missing or invalid configuration options
dependencies	an error during the downloading and resolution of dependencies
documentation generation	Javadoc and similar documentation errors
external program execution	communication with other programs/processes
file not found	a generic "file not found" error, not pertaining elsewhere
Java compilation	an error during the compilation of Java source code of the given project
non-Java compilation	an error during the compilation of non-Java source code (e.g., written in domain-specific languages [16]) of the given project
other build tool integration	communication with another build tool (e.g., Maven with Gradle)
packaging	error during the creation of the output archive
version control integration	errors related to Git and similar systems
wrong JDK version	the build tool itself explicitly mentioned an invalid Java version as the reason of the failure

3.3. Logs

Each build log is a text file containing both the standard output and standard error stream of the given build process, merged into one stream. The build tools were configured for batch mode via command-line switches (see Table 3) when possible. This means the logs should not contain unnecessary progress bars, ANSI color escape codes, etc.

In the case of Ant, we first executed the `clean` target as a separate process and then tried multiple common target names (`jar`, `war`, `dist`) before resorting to the default one (see Table 3). Therefore, every Ant log contains the output of 2 to 5 separate `ant` executions concatenated into one file. This is not a problem, though, since each `ant` execution begins with the output `Buildfile: .../build.xml` and ends with the line `Total time: ... seconds`. Thus, the file can be split into individual executions by these text patterns.

4. Data Summary

In this section, we provide basic descriptive statistics of the presented dataset. We particularly focus on the comparison between the years of execution.

4.1. Build Tools

First, we will look at the proportions of the detected main build tools in the projects. In Figure 1, we can see the majority of projects use Maven—and this proportion remains almost unchanged between the 2016 and 2020 execution (about 53%). The percentage of projects without a recognized build system stayed approximately constant, too (more than 27%).

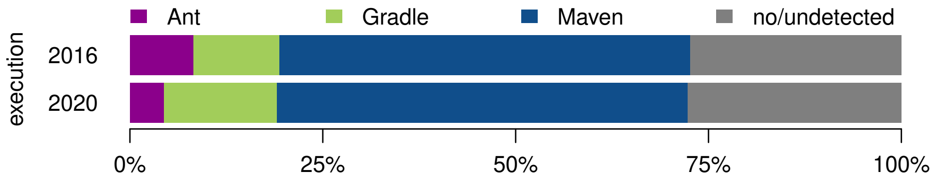


Figure 1. The percentages of the projects' recognized build tools.

On the other hand, the proportion of projects using Gradle increased from 11% to almost 15%. Ant, which can be considered almost legacy now, dropped from 8% to 4%.

4.2. Build Status

In the study run performed in 2016, the proportion of projects with a failing build was 38.1% (see Figure 2). In the 2020 execution, almost 59.4% projects failed to build from source. The portion of builds lasting longer than one hour, and thus timing out, is negligible for both execution years: approximately 0.1%.

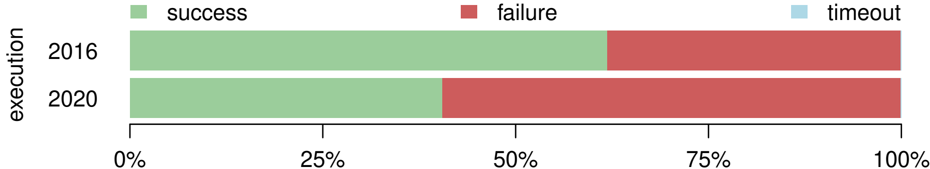


Figure 2. The percentages of the projects' build outcomes.

Figure 3 depicts the numbers of successful and failed builds divided by the last Git push year of the projects. We can see that a large portion of projects was updated relatively recently (with respect to the execution year) and very old projects are rare. The proportion of failed builds tends to be lower for more recently updated projects.

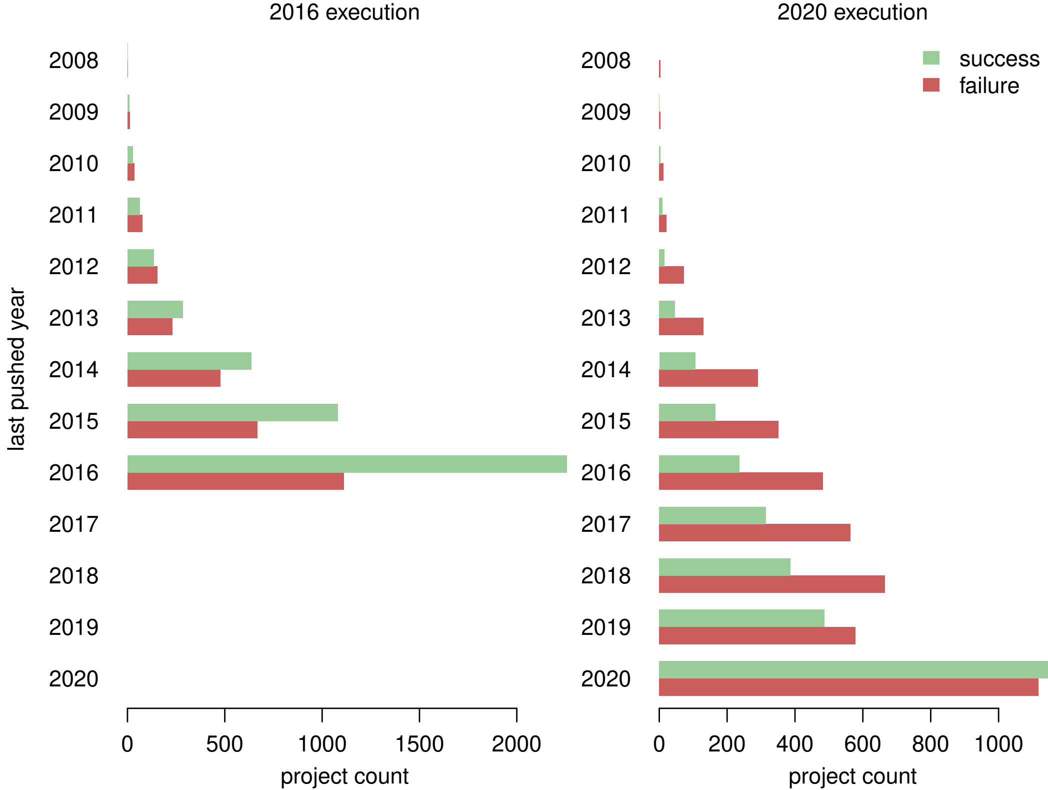


Figure 3. The numbers of successful and failed builds, divided by the last Git push year of projects.

4.3. Error Categories

Finally, in Table 6 we list the proportions of error categories for failing builds, as determined by our semi-automated analysis. We can see that while in the 2016 execution, the largest part of the errors was caused by dependency resolution and downloading, in the 2020 execution Java compilation errors were the most prevalent ones.

Table 6. Error categories of failed builds.

(a) The 2016 Execution	
Error Category	**%**
dependencies	38.93
Java compilation	22.54
uncategorized	8.72
documentation generation	6.4
configuration	4.85
build files parsing/interpretation	4.31
file not found	4.31
external program execution	2.35
packaging	2.35
version control integration	2.06
other build tool integration	1.63
wrong JDK version	0.83
non-Java compilation	0.72
(b) The 2020 Execution	
Error Category	**%**
Java compilation	36.23
dependencies	26.94
build files parsing/interpretation	11.13
documentation generation	7.64
uncategorized	6.26
configuration	6.01
file not found	1.23
version control integration	1.19
wrong JDK version	0.98
external program execution	0.88
other build tool integration	0.77
packaging	0.49
non-Java compilation	0.26

5. Potential Applications

The most obvious use of the published data is its more elaborate analysis. Since the execution from 2020 exhibits a large portion of failing builds, finding root causes of these errors is an important research direction. After a brief inspection of a small number of logs, we suspect backward-incompatible changes between Java 8 and Java 11 might cause some of the failures. However, this needs to be quantified and studied in more detail.

In our previous study [14], we found out there is a correlation between the build/success failure and the project's size, age, and update recency. There is potential in exploring the relationship between a build outcome and the rest of the available attributes.

Gallaba et al. [17] found that a small portion of Travis CI builds ignored an error—i.e., a failing build was recognized as a passing one. According to Ghaleb et al. [18], some builds fail due to temporary environmental issues, such as timeouts. Our dataset of local builds can be inspected with respect to these research problems, too.

Build breakage prediction [19,20] aims to detect whether the given build will fail as soon as possible, which minimizes the waste of developer's time and prevents potential timeouts. Both of the mentioned approaches utilize only data from CI builds; our dataset of purely local builds could conveniently complement them.

Since build error messages are notoriously difficult to comprehend, many researchers aim for better error explanation. For instance, BART [21] summarizes the reason of a failure and suggests possible fixes to the developer. Our dataset could be a valuable resource helping to design similar tools.

Build breakage repair [22–24] takes build fix suggestion a bit further—it aims to repair the build script fully automatically, without the developer's assistance. Our dataset of local build logs could be helpful for the improvement of similar approaches.

Overall, a higher quality of build systems may result in the improvement in many related research areas, ranging from studies of decompilers [25] which need compiled bytecode to proceed, to empirical program comprehension studies utilizing compilable projects as study objects.

Thanks to its relative simplicity, and since it contains both structured and unstructured data, our dataset can be also used in data science education as an example data set from the domain of software engineering. This could create a synergy between data science and software engineering education [26].

Since the complete source code used to produce the dataset is available (https://github.com/sulir/build-study), the study can be reused and modified with respect to the included set of projects, file patterns used for technology recognition, the execution environment, and analysis procedures.

Author Contributions: Conceptualization, M.S.; methodology, M.S.; software, M.S.; investigation, M.S.; data curation, M.B., M.M., M.S., S.C., and J.J.; writing—original draft preparation, M.S.; writing—review and editing, M.B., M.S., S.C., M.M., and J.J.; visualization, M.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by Project VEGA No. 1/0762/19 Interactive pattern-driven language development.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Neitsch, A.; Wong, K.; Godfrey, M. Build system issues in multilanguage software. In Proceedings of the 2012 28th IEEE International Conference on Software Maintenance (ICSM), Trento, Italy, 23–28 September 2012; pp. 140–149. [CrossRef]
2. Kerzazi, N.; Khomh, F.; Adams, B. Why Do Automated Builds Break? An Empirical Study. In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), Victoria, BC, Canada, 29 September–3 October 2014; pp. 41–50. [CrossRef]
3. Seo, H.; Sadowski, C.; Elbaum, S.; Aftandilian, E.; Bowdidge, R. Programmers' Build Errors: A Case Study (at Google). In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May 2014–7 June 2014; ACM: New York, NY, USA, 2014; pp. 724–734. [CrossRef]
4. Tufano, M.; Palomba, F.; Bavota, G.; Di Penta, M.; Oliveto, R.; De Lucia, A.; Poshyvanyk, D. There and back again: Can you compile that snapshot? *J. Softw. Evol. Process.* **2017**, *29*, e1838. [CrossRef]
5. Rabbani, N.; Harvey, M.S.; Saquif, S.; Gallaba, K.; McIntosh, S. Revisiting "Programmers' Build Errors" in the Visual Studio Context: A Replication Study Using IDE Interaction Traces. In Proceedings of the 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), Gothenburg, Sweden, 27 May–3 June 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 98–101. [CrossRef]
6. Horton, E.; Parnin, C. Gistable: Evaluating the Executability of Python Code Snippets on GitHub. In Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain, 23–29 September 2018; pp. 217–227. [CrossRef]
7. Rausch, T.; Hummer, W.; Leitner, P.; Schulte, S. An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software. In Proceedings of the 14th International Conference on Mining Software Repositories, MSR '17, Buenos Aires, Argentina, 20–28 May 2017; pp. 345–355. [CrossRef]
8. Zolfagharinia, M.; Adams, B.; Guéhéneuc, Y.G. A study of build inflation in 30 million CPAN builds on 13 Perl versions and 10 operating systems. *Empir. Softw. Eng.* **2019**, *24*, 3933–3971. [CrossRef]
9. Ghaleb, T.A.; Da Costa, D.A.; Zou, Y. An Empirical Study of the Long Duration of Continuous Integration Builds. *Empir. Softw. Eng.* **2019**, *24*, 2102–2139. [CrossRef]
10. Zhang, C.; Chen, B.; Chen, L.; Peng, X.; Zhao, W. A Large-Scale Empirical Study of Compiler Errors in Continuous Integration. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 12 August 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 176–187. [CrossRef]

11. Beller, M.; Gousios, G.; Zaidman, A. Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. In Proceedings of the 14th International Conference on Mining Software Repositories, MSR '17, Buenos Aires, Argentina, 20–28 May 2017; pp. 356–367. [CrossRef]
12. Madeyski, L.; Kawalerowicz, M. Continuous Defect Prediction: The Idea and a Related Dataset. In Proceedings of the 14th International Conference on Mining Software Repositories, MSR '17, Buenos Aires, Argentina, 20–28 May 2017; pp. 515–518. [CrossRef]
13. Brandt, C.E.; Panichella, A.; Zaidman, A.; Beller, M. LogChunks: A Data Set for Build Log Analysis. In Proceedings of the 17th International Conference on Mining Software Repositories, Seoul, Korea, 20–28 May 2017; Association for Computing Machinery: New York, NY, USA, 2020. [CrossRef]
14. Sulír, M.; Porubän, J. A Quantitative Study of Java Software Buildability. In Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools, Amsterdam, The Netherlands, 1 November 2016; ACM: New York, NY, USA, 2016; pp. 17–25. [CrossRef]
15. Spolsky, J. The Joel Test: 12 Steps to Better Code. In *Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity*; Apress: Berkeley, CA, USA, 2004; pp. 17–30. [CrossRef]
16. Kosar, T.; Oliveira, N.; Mernik, M.; Varanda Pereira, M.J.; Črepinšek, M.; Da Cruz, D.; Rangel Henriques, P. Comparing general-purpose and domain-specific languages: An empirical study. *Comput. Sci. Inf. Syst.* **2010**, *7*, 247–264. [CrossRef]
17. Gallaba, K.; Macho, C.; Pinzger, M.; McIntosh, S. Noise and Heterogeneity in Historical Build Data: An Empirical Study of Travis CI. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3 September 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 87–97. [CrossRef]
18. Ghaleb, T.A.; da Costa, D.A.; Zou, Y.; Hassan, A.E. Studying the Impact of Noises in Build Breakage Data. *IEEE Trans. Softw. Eng.* **2019**, 1–14. [CrossRef]
19. Hassan, F.; Wang, X. Change-Aware Build Prediction Model for Stall Avoidance in Continuous Integration. In Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '17, Markham, ON, Canada, 9–10 November 2017; pp. 157–162. [CrossRef]
20. Ni, A.; Li, M. Cost-Effective Build Outcome Prediction Using Cascaded Classifiers. In Proceedings of the 14th International Conference on Mining Software Repositories, MSR '17, Buenos Aires, Argentina, 20–28 May 2017; pp. 455–458. [CrossRef]
21. Vassallo, C.; Proksch, S.; Zemp, T.; Gall, H.C. Every build you break: developer-oriented assistance for build failure resolution. *Empir. Softw. Eng.* **2020**, *25*, 2218–2257. [CrossRef]
22. Hassan, F.; Mostafa, S.; Lam, E.S.L.; Wang, X. Automatic Building of Java Projects in Software Repositories: A Study on Feasibility and Challenges. In Proceedings of the 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Toronto, ON, Canada, 9–10 November 2017; pp. 38–47. [CrossRef]
23. Hassan, F.; Wang, X. HireBuild: An Automatic Approach to History-Driven Repair of Build Scripts. In Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 1078–1089. [CrossRef]
24. Macho, C.; McIntosh, S.; Pinzger, M. Automatically repairing dependency-related build breakage. In Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso, Italy, 20–23 March 2018; pp. 106–117. [CrossRef]
25. Kostelanský, J.; Dedera, L'. An evaluation of output from current Java bytecode decompilers: Is it Android which is responsible for such quality boost? In Proceedings of the 2017 Communication and Information Technologies (KIT), Vysoke Tatry, Slovakia, 4–6 October 2017; pp. 1–6. [CrossRef]
26. Luković, I. Issues and Lessons Learned in the Development of Academic Study Programs in Data Science. In *Data Analytics and Management in Data Intensive Domains*; DAMDID/RCDL 2019; Springer International Publishing: Cham, Switzerland, 2020; pp. 227–245. [CrossRef]

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).