LogChunks: A Data Set for Build Log Analysis

Carolin E. Brandt, Annibale Panichella, Andy Zaidman, Moritz Beller
{c.e.brandt,a.panichella,a.e.zaidman,m.m.beller}@tudelft.nl
Delft University of Technology
The Netherlands

ABSTRACT

Build logs are textual by-products that a software build process creates, often as part of its Continuous Integration (CI) pipeline. Build logs are a paramount source of information for developers when debugging into and understanding a build failure. Recently, attempts to partly automate this time-consuming, purely manual activity have come up, such as rule- or information-retrieval-based techniques.

We believe that having a common data set to compare different build log analysis techniques will advance the research area. It will ultimately increase our understanding of CI build failures. In this paper, we present *LogChunks*, a collection of 797 annotated Travis CI build logs from 80 GitHub repositories in 29 programming languages. For each build log, *LogChunks* contains a manually labeled log part (chunk) describing why the build failed. We externally validated the data set with the developers who caused the original build failure.

The width and depth of the *LogChunks* data set are intended to make it the default benchmark for automated build log analysis techniques.

KEYWORDS

CI, Build Log Analysis, Build Failure, Chunk Retrieval

ACM Reference Format:
Carolin E. Brandt, Annibale Panichella, Andy Zaidman, Moritz Beller. 2020. LogChunks: A Data Set for Build Log Analysis. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3379597.3387485

1 INTRODUCTION

Continuous Integration (CI) has become a common practice in software engineering [10]. Many software projects use CI [2, 10, 17] to detect bugs early [8, 18], improve developer productivity [10, 13] and communication [7]. CI builds produce logs which report results of various sub-steps within the build. These build logs contain a lot of valuable information for developers and researchers—for example, descriptions of compile errors or failed tests [2, 14, 20].

However, build logs can be verbose and large—sometimes in excess of 50 MB of ASCII text [2]—making them inadequate for direct human consumption. Therefore, to support developers and researchers in efficiently making use of the information within build logs, we must at least semi-automatically retrieve the chunks of the log that describe the targeted information.

There are different techniques to retrieve information chunks from CI build logs. Beller et al. use a rule-based system of regular expressions to analyze logs from Travis CI [2]. Such regular expressions are developed by looking at exemplary build logs. Vassallo et al. wrote a custom parser to gather information for build repair hints [19]. Recently, Amar et al. reduced the number of lines for a developer to inspect by creating a diff between logs from failed and successful builds [1].

These approaches have various strengths and weaknesses: Regular expressions are exact, but tedious and error-prone to maintain [12]. Custom parsers are powerful though fragile in light of changes in the log structure. Diffing between failed and successful logs can reduce the information to be processed, but is at best semi-automatic [1].

At the moment, there is only anecdotal evidence on the performance of these techniques, and when a technique should be preferred over its alternatives. In fact, there is no data set available to support the creation of such a benchmark for build log analysis techniques. Following Sim et al., a benchmark gives us the chance to "increase the scientific maturity of the area" [15] of build log analysis by evaluating and comparing research contributions.

Thus, in this paper, we present *LogChunks* [4],[1] a collection of 797 labeled Travis CI build logs from 80 highly popular GitHub repositories in 29 programming languages with we manually labeled the chunk describing why the build failed. The data set also provides keywords the authors would use to search for the labeled log chunk and categorizes the log chunks according to their format within the log.

2 CREATING *LOGCHUNKS*

This section presents how we gathered the logs and our manual labeling process.

2.1 Log Collection

In this section, we describe along the overview in Figure 1 how we created *LogChunks*. All steps are automatized as Ruby scripts[2] and highly configurable.

Repository Sampling. We target mature GitHub repositories that are using Travis CI. To avoid personal and toy projects we select popular projects based on the number of users that starred

[1] *LogChunks* is openly available on Zenodo: https://zenodo.org/record/3632351
[2] Data collection scripts and their original parameterization are included in the data set.

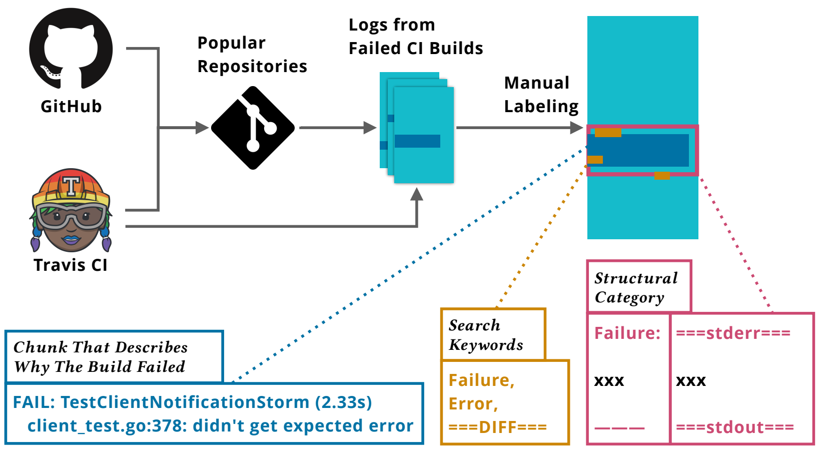
GitHub
Travis CI
Popular Repositories
Logs from Failed CI Builds
Manual Labeling
Chunk That Describes Why The Build Failed
FAIL: TestClientNotificationStorm (2.33s)
client_test.go:378: didn't get expected error
Search Keywords
Failure,
Error,
===DIFF===
Structural Category
Failure:
xxx
———
===stderr===
xxx
===stdout===

Figure 1: Overview of *LogChunks*

a project [11]. We query *GHTorrent* [9] for the most popular languages on GitHub, and subsequently, the most popular repositories for a given language.

For *LogChunks*, we queried GHTorrent from 2018-04-01 for the three most popular repositories of each of the 30 most popular languages to cover a broad range of development languages. Among the resulting repositories are, for example, `Microsoft/TypeScript`, `git/git` and `jwilm/alacritty`.

Build Sampling. To sample builds for *LogChunks* we keep the ten most recent builds of the status *failed* [6]. We check up to 1,000 builds per repository to ensure predictable termination of the log collection.

Log Sampling. Travis CI builds comprise a number of jobs that actualize a build process in different environments. Hence, the outcome from different jobs might be different. For each build in *LogChunks*, we download the log of the first job that has the same state as the overall build.

We inspected the collected build logs and discarded logs from three repositories. One had only one failed build, two others had empty build logs on Travis CI. In total, we collected 797 logs from 80 repositories spanning 29 languages.

2.2 Manual Labeling

After collecting build logs, the first author manually labeled which text chunk describes why the build failed. Following that, she assigned search keywords and structural categories to each log chunk.

Chunk That Describes Why The Build Failed. For each repository, the labeler skimmed through the build logs and copied out the first occurrence of a description why the build failed. She preserved whitespaces and special characters, as these might be crucial to detect the targeted substring. To support learning of regular expressions identifying the labeled substrings the labeler aimed to start and end the labeled substring at consistent locations around the fault description.

Search Keywords. To extract the search keywords, we considered the Chunk and ten lines above and below. The labeler's task was to note down three strings they would search for ("grep") to find this failure description. The strings should appear in or around the Chunk and are case-sensitive. We made no limitations on the search string; particularly, spaces are allowed.

Structural Category. To label the *structural categories* we presented the Chunk and the surrounding context to the labeler for all logs from a repository. We asked the labeler to assign numerical categories according to whether the Chunk had the same structural representation.

2.3 Validation

We validated our collected data points in an iterative fashion. First, we performed an initial inter-rater reliability study with the second author of this paper. Our learnings from this initial internal study are that 1) it is important and difficult to adequately communicate all decisions and assumptions on how to and which data to label and 2) there can be different legitimate viewpoints on which log chunk constitute the cardinal error and which keywords best to use. These learning informed the design of a second, larger cross-validation study for which we contacted over 200 developers.

In our second validation, we sent out emails to the original developers whose commits triggered the builds represented in *LogChunks* and asked them whether the log chunk we labeled actually describes why the build failed. This section describes our survey and discusses our results.

Method. Using the Travis API, we collected the commit information for each build represented in *LogChunks*. We grouped all commits triggered by one developer and sent out an email to them. It included links to the corresponding commits, the build overview and the log file. We asked the developers to fill out a short form in case our extraction was not correct. In the survey, we asked the developer to paste in the log part actually describing the failure reason or describe in their own words why our original extraction was incorrect.

Results. In total, from 2019-10-15 to 2019-10-17, we sent out emails to 246 developers. Of these, 32 could not be delivered. We performed the sending out in three batches and used the first author’s academic email address as the sender. All emails were specific to each recipient. We only sent one mail per recipient. We received answers from 61 developers, corresponding to 144 build logs with a response rate of 24.8%. Compared to typical response rates to cold calling known from Software Engineering [16], this is very high. We believe that our personalization and the ease of use for the participant are the main reasons for this—simply clicking on a link to confirm or refute an answer is enough, there is no need to craft an answer. Indeed, we only received seven replies from developers along the lines of “done.”

Of the 144 answers, 118 initially indicated that our extraction was correct. We manually inspected the 26 negative answers and found that some stated that the proposed extraction did not show the whole description of why the build failed. This was because we chose to trim long chunks to keep the mails readable, and not a fault in our extraction per se. After adjusting for these answers, only 12 answers remained that stated that our labeled log chunk was not correct. This validates our data set with an externally validated consensus on 94.4% of the extracted data.

Discussion. We believe that our developer survey highly strengthens the trust in the validity of the labeled log chunks. The study received answers for about 18% of the data in *LogChunks*. After manual correction, 91% of the received answers indicated our labeled chunks were accurate. One possible threat regarding the high number of correct answers is that, since we show the error message we extracted, it might be operationally easier for developers to validate it, rather than search for it in a long log file. To alleviate this problem, we made it as easy to confirm as to reject an extracted log chunk. We only ask for more details (the correct log chunk) in a second step.

One of our 12 incorrect extractions only showed a warning and the developer proposed to also include the line stating that warnings are treated as errors. In others, we labeled the error message of an error that was later ignored.

3 DATA SCHEMA

This section presents the internal structure and data schema of *LogChunks*. In principle, *LogChunks* comprises automatically retrieved and manually labeled and cross-validated logs.

LogChunks comprises information on 797 build logs, which are organized in folders for each language and repository. For each repository, *LogChunks* has about 10 `Examples`. Every repository folder contains the full log files for the build status ‘failed’ in plain text.

The folder `build-failure-reason` contains the manually labeled data of *LogChunks*, one XML file for each repository: `<repository_owner>@<repository_name>.xml`. Table 1 gives an overview of the data within these XML files on the example of one build from `php@php-src`. The remainder of this section defines in more detail the data embedded in the XML files, that is, the labeled log chunk, search keywords and structural categories. Data from the developer validation study is in the file

```
========DIFF========
-=-=-=-=-
005+     Parameter #1 [<optional> $flags]
005-     Parameter #1 [<optional> $ar_flags]
========DONE========
FAIL Bug#71412 ArrayIterator reflection
-=-=-=-=-
TEST 9895/13942 [2/2 test workers running]
```

Figure 2: Log chunk from the *same* structural category as the log chunk presented in Table 1. We inserted the special marker “-=-=-=-=-” to separate the log chunk from its context.

```
[0K$ ./sapi/cli/php run-tests.php -P ...
-=-=-=-=-
Illegal switch 'j' specified!
-=-=-=-=-
Synopsis:
```

Figure 3: Log chunk from a *different* structural category than the log chunk presented in Table 1.

`developer-crossvalidation.csv`, the build id can be used as a unique identifier to match it with the other data.

Chunk That Describes Why The Build Failed. The `Chunk` is the substring of the build log that describes why the build failed. This can, for example, be the failing test case or a compiler error. In cases where the reason *why* the build failed is contained in a log file external to the main build log, the `Chunk` includes only the fact *that* the build failed, for example “`The command "test/run !" exited with 1.`” In Table 1, the `Chunk` describes a failing test in which the tested process timed out.

Search Keywords. The `Keywords` contain a list of one to three freely chosen search strings appearing within the `Chunk` or in the area around it in the build log. We selected keywords the authors would use to search for the log `Chunk`, as we found them repeatedly next to failure describing chunks while analyzing about 800 build logs manually. Some keywords from *LogChunks* are “`Error`”, “`=DIFF=`”, “`ERR!`”, or the keywords shown in Table 1.

Structural Category. For each repository, we assign *structural categories* to the `Chunks`. The structural category compares how `Chunks` are represented within the build log. Build tools highlight their error messages with markings, e.g. starting each line with “`ERROR`” or surrounding special characters. Two chunks fall into the same structural categories if they are surrounded by the same markings. Listing 2 presents a log chunk from the same category as the log chunk from Table 1. In comparison to that, Listing 3 presents a log chunk which is formatted differently within the log file. For each repository, the structural categories are represented as integers, starting at 0 and increased with the next appearing category in chronological build order.

4 POTENTIAL USE CASES

LogChunks can be the basis for a range of further studies:

Benchmarking Build Log Analysis Techniques. LogChunks originated from the first author’s Master’s Thesis in which she compared three different log chunk retrieval techniques. *LogChunks* can be

Data	Description	Unit	Example
Log	Relative path to the input build log	Unique Path	`C/php@php-src/failed/529279089.log`
Chunk	Log chunk that describes why the build failed	String	`001+ ** ERROR: process timed out **`
			`001- OK.`
			`========DONE========`
			`FAIL Bug #60120 (proc_open hangs)`
Keywords	Keywords the authors would use to search for and find the log chunk	List of Strings	`ERROR, FAIL, DIFF`
Category	Categorization of the structural representation of the log chunk within the build log	Integer	`0`

Table 1: Exemplary, complete data excerpt from *LogChunks* for a failed build in `php@php-src`.

a benchmark to evaluate other build log analysis techniques. For example, one can use the data set to investigate how reliably the diff-based approach of Amar et al. [1] retrieves build failure reasons.

Support Build Log Classification Algorithms. Various researchers examine why CI builds fail and use build logs as a data source [14, 20]. They typically write custom parsers and classifiers to categorize builds according to why a build failed. The manually labeled chunk can help researchers locate the source for their classification algorithms and cross-validate their data.

Research on Build Logs. The data from *LogChunks* can support research around the topic of build logs such as how developers use keywords to retrieve information about build failures from logs or how they discuss failures of CI builds within pull requests [5].

Automatic Processing of Build Results. LogChunks enables researchers to train algorithms that retrieve build failure descriptions from build logs. It can provide the basis for further automatic on-ward processing of the retrieved log chunks.

5 RELATED DATA SETS

This section presents existing data sets of CI build logs and how *LogChunks* differs from them.

5.1 TravisTorrent

TravisTorrent [3] collects a broad range of metadata about Travis CI builds. It combines data accessible through the public Travis CI and GitHub APIs and through GHTorrent [9]. Similar to *LogChunks*, among the metadata are the failing test cases. However, TravisTorrent obtained these through a manually developed parser, which only supports specific Ruby test runners and Java Maven or JUnit logs. Anecdotally, many of the failing tests are at least incomplete and lack validation. By contrast, *LogChunks* provides manually labeled and two-fold cross-validated data of why builds failed, not only for failing tests like TravisTorrent, but for all possible build-failing errors.

5.2 LogHub

LogHub [21] is a collection of a wide range of system log data sets. It is the basis for various studies that compare different approaches to parse unstructured system log messages into structured data for further analysis. *LogChunks* is situated in a different area, build log analysis, which tend to be semi-structured, and could play a similar role to *LogHub* in its area: empower research with a benchmark to compare different build log analysis techniques.

6 FUTURE EXTENSIONS TO *LOGCHUNKS*

In this section, we describe current limitations and future improvements of *LogChunks* and extensions we are planning.

Chunk as One Consecutive Substring. The `Chunk` contains only one continuous substring of the log text. The reason a build failed could be described at multiple locations within the log. We propose to extend *LogChunks* to contain multiple substrings of the log text.

Include More Repositories and Logs. LogChunks encompasses a range of repositories from various main development languages, though only 10 logs from each repository. Including more logs and repositories will strengthen *LogChunks* as the go-to benchmark.

Classification of the Build Failure Cause. Our data set contains no further classification according to the cause of the failure, such as due to tests, compilation or linter errors. As researchers are investigating why CI builds fail, a useful extension is to annotate cause of the build failure for each log.

Other Information Chunks. Build log analysis is not limited to the chunk that describes why a build failed. *LogChunks* can be extended with labels for all information that is contained in the build log, such as descriptions of warnings, build infrastructure and more.

Validation of Search Keywords. The keywords *LogChunks* provides are based on the experience of the authors gained from analyzing around 800 build logs. Next, we propose to evaluate whether these keywords would also be used by developers to find the log chunk describing why a build failed.

7 SUMMARY

In this paper, we introduce *LogChunks*, a cross-validated data set encompassing 797 build logs from 80 projects using Travis CI. For each log, we annotated the log chunk describing why the build failed and provided keywords a developer would use to search for the log chunk as well as a categorization of the log chunks according to their format within the log. *LogChunks* advances the research field of build log analysis by introducing a benchmark to rigorously examine research contributions [15] and opening various research possibilities that previously required tedious manual classification.

REFERENCES

[1] Anunay Amar and Peter C Rigby. 2019. Mining Historical Test Logs to Predict Bugs and Localize Faults in the Test Logs. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE, 140–151.

[2] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. In *Proceedings of the 14th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE, 356–367.

[3] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proceedings of the 14th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE, 447–450.

[4] Carolin Brandt, Annibale Panichella, and Moritz Beller. 2020. LogChunks: A Data Set for Build Log Analysis. https://doi.org/10.5281/zenodo.3632351

[5] Nathan Cassee, Bogdan Vasilescu, and Alexander Serebrenik. 2019. The Silent Helper: The Impact of Continuous Integration on Code Reviews. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society.

[6] Travis CI. 2019. Travis Build Status. https://docs.travis-ci.com/user/job-lifecycle/#breaking-the-build. Accessed: 2019-11-18.

[7] John Downs, Beryl Plimmer, and John G Hosking. 2012. Ambient Awareness of Build Status in Collocated Software Teams. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. IEEE, 507–517.

[8] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education.

[9] Georgios Gousios. 2013. The Ghtorrent Dataset and Tool Suite. In *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 233–236.

[10] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*. ACM, 426–437.

[11] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2016. An In-Depth Study of the Promises and Perils of Mining GitHub. *Empirical Software Engineering* 21, 5 (2016), 2035–2071.

[12] Louis G Michael IV, James Donohue, James C Davis, Dongyoon Lee, and Francisco Servant. 2019. Regexes Are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[13] Ade Miller. 2008. A Hundred Days of Continuous Integration. In *Agile 2008 Conference*. IEEE, 289–293.

[14] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' Build Errors: A Case Study (At Google). In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, 724–734.

[15] Susan Elliott Sim, Steve Easterbrook, and Richard C Holt. 2003. Using Benchmarking to Advance Research: A Challenge to Software Engineering. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*. IEEE, 74–83.

[16] Edward Smith, Robert Loftin, Emerson Murphy-Hill, Christian Bird, and Thomas Zimmermann. 2013. Improving developer participation rates in surveys. In *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 89–92.

[17] Daniel Ståhl and Jan Bosch. 2014. Modeling Continuous Integration Practice Differences in Industry Software Development. *Journal of Systems and Software* 87 (2014), 48–59.

[18] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 805–816.

[19] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C Gall. 2018. Un-Break My Build: Assisting Developers with Build Repair Hints. In *Proceedings of the 26th IEEE International Conference on Program Comprehension (ICPC)*. ACM, 41–51.

[20] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. 2017. A Tale of CI Build Failures: An Open Source and a Financial Organization Perspective. In *Proceedings of the 33th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 183–193.

[21] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. 2019. Tools and Benchmarks for Automated Log Parsing. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 121–130.