



SOFTWARE ENGINEERING

Elite Graduate Program

Master's Thesis

How to Analyze Build Logs

—

A Comparative Study of Chunk Retrieval Techniques

Carolin E. Brandt



Institut für Software & Systems Engineering

Universitätsstraße 6a D-86135 Augsburg



SOFTWARE ENGINEERING

Elite Graduate Program

Master's Thesis

How to Analyze Build Logs

—

A Comparative Study of Chunk Retrieval Techniques

Autorin: Carolin E. Brandt

Beginn der Arbeit: 5. November 2019

Ende der Arbeit: 5. Mai 2020

Erstgutachter: Prof. Dr. Alexander Pretschner

Zweitgutachter: Prof. Dr. Alexander Knapp

Betreuer: Dr. Moritz Beller

Prof. Dr. Annibale Panichella



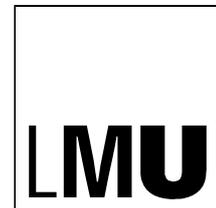
Delft
University of
Technology



TECHNISCHE
UNIVERSITÄT
MÜNCHEN



Universität
Augsburg
University



Institut für Software & Systems Engineering

Universitätsstraße 6a D-86135 Augsburg

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Augsburg, den 7. Januar 2020

Carolin E. Brandt

Acknowledgments

I would like to thank Dr. Moritz Beller for his great support, his detailed feedback and his invaluable teaching on how to do research. I could not have imagined a better advisor for my master's thesis.

Furthermore, I would like to thank Prof. Annibale Panichella for his helpful feedback and teaching me the proper vocabulary to describe my work.

I want to pay my special regards to Prof. Alexander Pretschner for his critical questions, leading me to continuously challenge my evaluation and the problems my work addresses.

I would like to thank Prof. Alexander Knapp for his sharp eyes for formulas and our reassuring conversations.

I would like to acknowledge Martin, Martijn and Dominik for their thorough reviews.

I am extremely grateful for my newfound friends, the fourth floor group and my colleagues from SERG. Thank you for welcoming me with open arms and sparking joy in my days.

Lastly, I want to thank Michi and Taico. Thank you for your calming input, your critical reflection, your relentless support and your upcheering.

Abstract

Continuous integration produces detailed logs about the status and results of the various tools involved in the build. These build logs are a valuable data source for developers and researchers to inspect test results, to check the duration of build steps and to understand the cause of a build failure. However, build logs are very verbose, at best semi-structured and their structure differs highly between projects. This makes it hard to process and analyze them. In this thesis, we evaluate and compare three different techniques that aim to retrieve specified log parts (chunks) from a build log, namely program synthesis by example, textual similarity and search keywords. We conduct an empirical study by comparing these techniques on our manually labeled *LogChunks* data set of 797 Travis CI build logs from a broad range of 80 projects. Our findings show that none of the three techniques in general outperforms the others. We discuss under which circumstances each technique performs best and provide a recommendation on when developers or researchers should use which technique.

Contents

1	Introduction	1
2	Background and Related Work	5
2.1	Continuous Integration and Build Logs	5
2.1.1	Continuous Integration	5
2.1.2	Augmentation of Build Logs	7
2.1.3	Build Logs as Semi-Structured Data	8
2.1.4	System Log Analysis	9
2.2	Foundational Techniques	9
2.2.1	Program Synthesis by Example	10
2.2.2	Information Extraction	12
2.2.3	Information Retrieval	13
3	Chunk Retrieval Techniques for Build Logs	15
3.1	Characteristics of a Build Log	15
3.2	Information Chunks in Build Logs	16
3.3	Characteristics of Chunk Retrieval Techniques	18
3.3.1	Program Synthesis by Example (PBE)	21
3.3.2	Common Text Similarity (CTS)	21
3.3.3	Keyword Search (KWS)	22
3.3.4	Other Techniques	22
3.4	Tool Implementation	23
4	LogChunks Data Set	25
4.1	Motivation	25
4.2	Related Data Sets	27
4.3	Data Schema	27
4.4	Log Collection	29
4.5	Labeling Process	30
4.6	Validation	31
4.6.1	Inter-Rater Reliability Study	31
4.6.2	Developer Survey	32
5	Empirical Comparison	37
5.1	Study Design	37
5.2	Results	39
5.2.1	Program Synthesis by Example (PBE)	39
5.2.2	Common Text Similarity (CTS)	40
5.2.3	Keyword Search (KWS)	40
5.2.4	Comparison of All Techniques	41
6	Discussion	49

6.1	Interpretation of Study Results	49
6.1.1	Program Synthesis by Example (PBE)	50
6.1.2	Common Text Similarity (CTS)	50
6.1.3	Keyword Search (KWS)	50
6.2	Recommendations of Suitable Techniques	51
6.3	Threats to Validity	52
7	Conclusion and Future Work	55
	Bibliography	59

1 Introduction

Continuous Integration (CI) has become a common practice in software engineering [1]. Many software projects use CI [1–3] to detect defects early [4, 5] and to improve developer productivity [1, 6] and communication [7]. A build on a CI server typically compiles and packages the software, executes tests [3] and performs various kinds of static analyses [8].

CI builds produce logs which report the results of the steps within the build. These build logs contain valuable information for developers and researchers. Developers examine log statements of time measurements to track how steps of the build process perform. Researchers collect the commands triggering different build steps to reverse engineer the build configuration when only the log is available to them. Both groups analyze build logs for descriptions of compiler errors or failed tests to uncover why a CI build failed [3, 9, 10].

The problem is, that build logs are verbose and long [3], making them inadequate for direct consumption. Therefore, developers and researchers can only efficiently use the information within build logs if they can adequately retrieve those parts (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 regular expressions to analyze the reasons of build failures from Travis CI logs [3]. Vassallo et al. wrote a custom parser for build logs to gather information for build repair hints [11]. Recently, Amar et al. reduced the number of build log lines for a developer to inspect by creating a diff between the logs from a failed and a successful build [12].

These approaches have various strengths and weaknesses: Regular expressions are exact but difficult to maintain. They are developed by looking at a few exemplary build logs. Updating regular expressions whenever new cases are introduced is a tedious and error-prone task [13]. Custom parsers are powerful though fragile towards changes in the log structure. The structure of build logs changes from project to project [2] as it depends on the tools and the environment used. Taking a diff between the logs of failed and successful builds can reduce the information to be processed, but the imprecise output needs to be interpreted by a developer [12].

At the moment there is only anecdotal evidence on the performance of these techniques and on when a technique should be preferred over other alternatives. Developers and researchers currently have little support when choosing which technique to use for a task.

The goal of this thesis is to investigate different chunk retrieval techniques for build logs and describe under which circumstances certain techniques can be recommended over others. We aim to characterize different chunk retrieval techniques, as well as the information retrievable from CI build logs. For **Research Question 1 (RQ1)**, we analyze which criteria influence the suitability of a chunk retrieval technique for CI build logs.

We implement and evaluate three chunk retrieval techniques:

- program synthesis by example using the Microsoft PROSE library (referred to as PBE),
- a common text similarity approach (referred to as CTS), and
- keyword search (referred to as KWS).

RQ2 asks under which conditions PBE, CTS and KWS are suited to retrieve information from continuous integration build logs. **RQ2** is refined into sub-questions along with the criteria resulting from **RQ1** and compares their instantiations for the three techniques: how many training examples a technique needs to perform best (**RQ2.1**), how structurally diverse the examples can be (**RQ2.2**) and how accurate the retrieved output is (**RQ2.3**). To evaluate PBE, CTS and KWS we create a data set, called *LogChunks*, which encompasses about 800 log files from 80 repositories. Each log is labeled with the log part describing the reason a build failed, keywords to search for this log part and a categorization of the labeled log part according to its structural representation within the log.

Our study of the three techniques on *LogChunks* shows that

- PBE yields very accurate results when trained with two examples from a single structural category.
- CTS shows the best average precision, though precision and recall of a retrieval is hard to determine from the given result. A small increase in the number of training examples has no noticeable influence. Fewer structural categories improve precision and recall of the retrieval.
- KWS has the highest recall of all techniques, however much lower precision. It is the technique with the best recall when multiple structural categories are present in the training examples.

We recommend PBE for use cases where the desired information is always represented in the same structural way and high confidence in precision and recall of the chunk retrieval is required. CTS is well suited when the representation of the desired information varies slightly and the output of the chunk retrieval is further processed by a human. In cases where the textual representation of the desired information in the log is unpredictable or varies greatly, KWS is the best technique to choose. However, its low precision requires a human to interpret the output of the chunk retrieval.

Our work contributes:

- A tool unifying several chunk retrieval techniques namely:
 - program synthesis from examples using the Microsoft PROSE library (PBE),
 - a common information retrieval approach using text similarity (CTS), and
 - a keyword search approach (KWS).
- A validated data set of about 800 logs from failed Travis CI builds manually-labelled with:
 - the substring of the log describing the reason the build failed,
 - keywords we would use to search for these substrings, and
 - a categorization of the substrings according to their structural representation within the build log.
- Recommendations for the configuration of each of the investigated chunk retrieval techniques.
- Guidelines on choosing a suitable chunk retrieval technique.

This thesis first presents an overview of related research, spanning from CI, build log analysis and augmentation to system log processing. Chapter 2 also presents existing work on information extraction and retrieval techniques, as well as program synthesis from examples. Next, Chapter 3 characterizes chunk retrieval techniques and retrievable information from CI build logs. It also introduces the three investigated techniques PBE, CTS and KWS. Chapter 4 describes the creation of the *LogChunks* data set collected from failed Travis CI build logs, including the labeling and the validation process. Chapter 5 explains the empirical comparison of the three techniques on *LogChunks* and presents the study results. Chapter 6 discusses the implications of the study results and recommendations on when PBE, CTS and KWS are most suitable. Lastly, Chapter 7 concludes and gives an overview of further research opportunities.

2 Background and Related Work

This chapter presents various research works adjacent or foundational to our work. The first section presents existing work in the field of continuous integration (CI) and showcases how researchers analyze build logs. In the second section we describe the existing methods which are the foundation for the chunk retrieval techniques we investigate.

2.1 Continuous Integration and Build Logs

We describe existing studies about CI to show that CI is a relevant topic within the software engineering research community. We explain how researchers gather information about CI usage in software projects through build log analysis and why the chunk retrieval techniques we investigate simplifies their data collection. This section moves on to past works about augmenting build logs to make it easier for developers to inspect them. The presented approaches are related to and can benefit from the techniques we analyze in this thesis. Further, we classify build logs as semi-structured data and differentiate our work from system log analysis.

2.1.1 Continuous Integration

The first paragraphs of this section present the results of past research into CI: its influence on the software engineering processes, the reasons for CI builds to fail and the role of testing and of static analysis in CI builds. Later we describe how researchers obtain the data sets these works are based on. We show how they analyze build logs and why the chunk retrieval techniques we investigate can support them to broaden their studies.

Motivation for CI and Impact of CI Why teams choose to use CI and its impact on the software development process is explored by several existing works. Hilton et al. [14] investigate the motivations of developers to use CI through several surveys. They find that, developers use CI to ensure consistency and quality across different execution environments and increase confidence in the code they deploy. Hilton et al. [1] also analyze how and why open source projects use CI. They observe CI usage in a broad range of projects, it supports developers to catch bugs earlier and to shorten release cycles. Stahl and Bosch [2] provide a review of literature on automatic build environments in industry projects. They propose a descriptive structure to model build flows, which they found to be highly different from project to project. Vasilescu et al. [4] analyze a broad range of open source projects written in popular languages on GitHub. They compare the usage of CI with the successful merges of pull requests and find that CI increase the number of successful merges and allows the team members to uncover more bugs. In a recent study, Vasallo et al. [15] interview developers on how they determine why a CI build failed and model how these developers resolve failures. Vasallo et al. find that, the first and most important steps when developers want to fix a build failure is to locate the error within the build log. Then

the developers use the additional information provided with and around the error message to understand the details of the failures. This shows that the build log is a central source of information about a CI build.

Build Failures in CI Various researchers look into why CI builds fail and into the impact of build failures on the development workflow. Seo et al. [9] find that a small group of error types such as dependency mismatches are the most prominent cause of build failures at Google. In addition, they notice that most failures are resolved within two builds. Rausch et al. [16] analyze CI builds of open source Java projects and find that most builds fail because of failing tests. For most projects, over half of the failed builds follow a previous failed build. Rausch et al.’s data shows that most failures occur in the second half of the build runtime, which can cause long delays in the feedback loop, especially when builds are automatically retried upon failure. Vassallo et al. [10] compare open source projects in Java to industrial ones. They determine that testing failures are more common than compilation errors. Open source builds fail most often because of unit tests, whereas release preparations are the primary cause in industrial projects.

Static Analysis and Testing in CI Automated static analysis tools are the focus of Zampetti et al. [8] in their study of Java projects from GitHub. Their results show that static analysis is responsible for a small amount of build failures and mainly responsible for warned builds. Almost all analyzed projects use custom configurations for the static analysis, however the configuration rarely changes. Failures because of static analysis warnings are observed to be fixed in a short time frame. Beller et al. [3] show that testing is central to continuous integration when evaluating Travis CI logs for Java and Ruby builds. They observe a high variation between programming languages, in the kind and number of tests which are run, as well as how often tests fail. The low failure rates on the CI server hint at code being tested before it is sent to the server.

Data Sets Powering CI Research The works presented in this section are based on either developer surveys or data sets containing build metadata or build logs. Seo et al. [9] and Vassallo et al. [10] based their analyses on sets of build logs collected from industry partners. Beller et al. [17] created the *TravisTorrent* data set providing access to build metadata from more than 1,000 projects from Travis CI [18]. *TravisTorrent* was the basis for several of the works mentioned in this section [3, 8, 10, 16]. Ghaleb et al. [19] aim to identify noise in build breakage data. They classify build failures from *TravisTorrent* according to whether they were caused by an environmental failure or caused by a developer change. They also identify cascading build failures created by existing unfixed errors and allowed failures, whose results were later labeled by developers to be ignored. About half of the failed builds in *TravisTorrent* fall into at least one of these categories. When regarded as noise they considerably impact observations reported by other works modeling build breakages.

Build Log Analysis in CI Research Several of the works we mentioned until now analyze build logs to obtain information about the CI builds. For Seo et al. [9] the build logs are the primary data source for their study. They develop a custom parser to classify error messages reported by Java and C++ builds. Vassallo et al. [10] analyze collected build logs by extracting error messages using regular expressions. The regular expressions search for keywords identified in a manual analysis. The analysis of Ghaleb et al. [19] starts with manual categorization of build logs. They select keywords and strings that identify their targeted categories and code a script to automatically classify logs based on these keywords. Beller et al. [3] focus their analysis on Java and Ruby build logs for which they build custom parsers with regular expressions to extract the reason a build failed.

Supporting Log Analysis with Chunk Retrieval To leverage the valuable information within build logs the researchers presented in this section build parsers and regular expression-based programs. This task of retrieving specific chunks of text from the build logs can be solved by the chunk retrieval techniques we compare in this thesis. Our results can support researchers in choosing a suitable technique for their data set of build logs and the chunks they want to retrieve. By relieving them from building custom parsers we enable them to cover a much wider range of languages and build tools in their studies.

2.1.2 Augmentation of Build Logs

Build logs are a valuable data source for developers to find out why their build failed. Several researchers are looking into supporting developers to process the verbose build logs. Vassallo et al. [11] try to shorten the time it takes developers to understand build logs. They parse Maven [20] build logs into a structured representation and create hint generators. The hint generators leverage this structured access to the information within the build log to propose fixes. For example, one of the hint generators queries stack overflow for discussions related to why the build failed. In a qualitative study they observed that highlighting the locality and context of an issue is helpful to programmers. Their tool BART is published as a Jenkins Plugin [21]. The chunk retrieval techniques we compare in this thesis can be used to fill similar structured representations with information from build logs. As they simplify the construction of parsers they would enable developers and researchers to cover a wider array of build tools, which is the main influence factor on the structure of a build log.

Amar et al. [12] compare different approaches to reduce the portions of a log that a developer has to inspect. Their techniques remove lines that appear both in logs from passing and failing builds and use a modified *Term Frequency Inverse Document Frequency* (TF-IDF) weighting to identify term vectors likely to occur with failures. Their diff technique can be interpreted as a chunk retrieval technique, where the targeted information is defined by the past failures used as basis for the weighted term vectors.

Travis itself already provides build log augmentation in the form of a basic structuring of the build logs within their log viewer using *log folds* [22]. They add fold identifiers around common commands and setup or teardown build phases and collapse the contained lines by default.

2.1.3 Build Logs as Semi-Structured Data

Serge Abiteboul introduces a theory of semi-structured data in his report from 1997 about integrating data from several sources [23]. He proposes essential characteristics of semi-structured data. In this section, we show how five of these map to the context of build logs.

The structure of build logs

- **is implicit.** We might not have access to explicit structuring elements or an explicit structure description. Computation is required to infer the present structure.
- **is irregular.** Changes in the build process or execution environment might change the structure of a build log. We observed this in the logs we collected for the *LogChunks* data set, where for the same repository and build configuration some logs had double new line characters without any noticeable explanation. Figure 2.1 and Figure 2.2 present an example of this.
- **is partial.** Some parts are highly structured by e.g. special characters. Other parts are unstructured, such as natural language text in error messages.
- can be described with an **analytical data guide** rather than a predefined schema. There is no fixed specification on how build tools structure their output. We later extract structuring patterns in the produced output.
- **has a rapidly evolving schema.** Modifications in the build configuration can change the tools involved in the build and therefore the composition of the build log. Previous work indicates that some projects change their CI configuration often [1] and software tools adapt their log messages over time [24].

Abiteboul proposes an approach to overlay the semi-structured data with a structured layer. The additional layer can answer queries and give access to the semi-structured data relevant to a query. In contrast to that, this thesis takes a look at techniques to gather a specific, pre-specified information without the need to parse, understand or estimate the whole structure of a log.

```
!Unit/DynamicDictionaryFixture.cs(1147,13): warning xUnit2007: Do not use typeof(string) expression to check the type
!Unit/ModelBinding/DefaultConverters/FallbackConverterFixture.cs(64,63): warning xUnit1026: Theory method 'Should_con
!Unit/NamedPipelineBaseFixture.cs(235,13): warning xUnit2013: Do not use Assert.Equal() to check for collection size.
y.Tests -> /Users/travis/build/NancyFx/Nancy/test/Nancy.Tests/bin/Release/netcoreapp2.0
WarningMessage
```

Figure 2.1: Excerpt from a build log showing a *WarningMessage* chunk.

```
!Unit/DynamicDictionaryFixture.cs(1147,13): warning xUnit2007: Do not use typeof(string) expression to check the type
!Unit/ModelBinding/DefaultConverters/FallbackConverterFixture.cs(64,63): warning xUnit1026: Theory method 'Should_con
!Unit/NamedPipelineBaseFixture.cs(235,13): warning xUnit2013: Do not use Assert.Equal() to check for collection size.
-> /home/travis/build/NancyFx/Nancy/test/Nancy.Tests/bin/Release/netcoreapp2.0
WarningMessage
```

Figure 2.2: Excerpt from a build log showing a differently formatted *WarningMessage* chunk.

```

1 2008-11-09 20:46:55,556 INFO dfs.DataNode$Packet -
   ↳ Responder: Received block blk_3587508140051953248 of
   ↳ size 67108864 from /10.251.42.84
2 2008-11-09 20:49:46,764 WARN PacketResponder 0 for block
   ↳ blk\_3587508140051953248 terminating

```

Figure 2.3: *System Log Statements. Example adapted from [25].*

2.1.4 System Log Analysis

A related field of log processing is the processing of system log files produced during runtime. A main difference between build logs and system logs is that system logs are fundamentally structured through events. Each line in a log file represents one event with a set of fields: timestamp, verbosity level and raw message content [25]. Figure 2.3 shows example lines from a system log.

The first goal in parsing system log files is to separate constant and variable parts within a log message [25, 26]. Next, the log messages are clustered into log events, unifying messages with identical constant parts and varying parameters. The output of a log parser is a structured log, composed of a list of timed events and the corresponding parameter values [27]. This structured log is then the input to various machine learning and data mining processes. Researchers mine patterns for operational profiling [28], debugging [29], performance analytics or anomaly detection [26]. Xu et al. [30] leverage the connection of log statements to the source code producing them to separate messages into constant and variable parts more accurately.

The techniques developed for system log analysis can also be applied to build logs. One example is comparing execution traces to reference traces of intended behavior to detect anomalies. Amar et al. [12] employed a similar approach to detect relevant lines in build logs.

Classic log parsers interpret the whole log file into a sequence of events. A similar approach could also be applied to build logs to determine the sequence of executed build steps or phases. In this thesis we take a different approach and focus on extracting a single specified information from the build log as a whole with chunk retrieval techniques. Chunk retrieval techniques are used as a part of log parsing to retrieve the values of variable parts in a log message, e.g. by using regular expressions [26, 30].

2.2 Foundational Techniques

The techniques we investigate are based on existing methods of Programming by Example, information extraction and information retrieval. This section presents different Programming by Example resources surrounding the PROSE library. We explain its generic program synthesis algorithm and how PROSE synthesizes text extraction programs, the foundation of PBE. In addition, we describe how chunk retrieval can be employed as a part of presented information extraction approaches. We introduce how information retrieval techniques, the basis for CTS, are used to improve software development.

2.2.1 Program Synthesis by Example

Programming by Example enables end users to automate repetitive tasks. The user provides examples for the input and the corresponding output and a synthesis algorithm tries to create the program intended by the user. This section introduces the theoretical foundations of the program synthesis algorithm of the *PROgram Synthesis using Examples* (PROSE) framework [31]. The PROSE framework is developed by Microsoft Research. Next, this section presents the FlashExtract DSL, which defines text extraction tasks within PROSE and is the basis for the implementation of our chunk retrieval technique PBE. Finally, we cover additional applications and extensions of the PROSE program synthesis and alternative research on generating regular expressions from examples.

FlashMeta: Inductive Program Synthesis The FlashMeta framework presented by Polozov and Gulwani [32] is the backbone of the program synthesis in the Microsoft PROSE framework. FlashMeta separates the inductive synthesis algorithm from the domain specific capabilities of the desired program by encoding the possible program space in a domain specific language (DSL). The user specifies the desired program behavior by providing in/output examples (I/O examples). FlashMeta uses *witness functions*, provided by the DSL, to divide the synthesis into smaller subtasks. For each of these subtasks it enumerates all possible programs that solve the subtask consistent with the set of I/O examples. A program is consistent with a set of I/O examples if, for each input example, it produces the corresponding output [33]. The possible subprograms are joined and stored in a *version space algebra* (VSA) [33]. This a tree structure, which space-efficiently saves candidate programs for tasks by sharing common subexpressions. Next, FlashMeta ranks the enumerated programs according to which ones the user most likely intended. The DSL also provides the ranking characteristics. From the ranked VSA, FlashMeta can then return a ranked list of complete programs consistent with the user’s example.

In addition to I/O examples of the intended program, the user can also provide examples with only input or negative input examples. Negative input examples should not be processed by the synthesized program.

The different applications of PROSE presented in the following paragraphs were all eventually implemented as DSLs for the FlashMeta synthesis algorithm.

FlashExtract: Data Extraction by Example Le et al. [34] developed FlashExtract as a DSL for the Microsoft PROSE framework. It enables a user to define text extraction programs for text, websites and spreadsheets by giving I/O examples. FlashExtract’s instantiation for text synthesizes extraction programs from semi-structured text based on regular expressions. Users can extract multiple fields and structure them with hierarchy and sequence. FlashExtract synthesizes programs to extract each of the fields leveraging the information about hierarchical containment and sequentiality. It eliminates the need for the user to understand the entire structure of the processed document and decreases the effort of developing a suitable extraction program.

The text instantiation of FlashExtract models the extraction of a single substring as a pair of two cut positions. A position is either specified by an absolute character index or by

```

1 let s = v in let s = PosToEndRegion(s, RegexPosition(s,
  ↳ RegexPair("Colon◦Line Separator", "ALL CAPS"), 1)) in
  ↳ StartToPosRegion(s, RegexPosition(s, RegexPair("ε",
  ↳ "Line Separator◦Line Separator"), 1))

```

Figure 2.4: Text extraction program synthesized by FlashExtract.

a pair of two regular expressions. The first regular expression matches the substring directly before the characterized position, the second regular expression matches the substring directly after. A regular expression in FlashExtract is a concatenation of tokens, e.g. standard character classes or string literals frequently occurring in the input examples. Figure 2.4 shows a text extraction program synthesized by FlashExtract. This program defines the first position as after a colon followed by a newline character and before a piece of text with all capital letters. It defines the second position as before two newline characters.

Apart from automatic completion in Excel spreadsheets [35], FlashExtract is the basis for two other Microsoft product features: Microsoft’s system log analysis tool Azure Monitor lets users define custom log fields [36]. The ConvertFrom-String function in PowerShell allows a user to specify an example template to extract hierarchical data from a text document [37].

We apply the text instantiation of FlashExtract to the domain of build logs with our chunk retrieval technique PBE.

Other Applications of Program Synthesis by Example Gulwani and Harris apply a less generic predecessor of the FlashMeta framework to string manipulation within spreadsheets [38] and spreadsheet transformations [39].

Rolim et al. [40] use code edits as examples to learn automatic program transformations. Their DSL for PROSE abstracts over variables and subexpressions by describing rewrite rules applied to the abstract syntax tree. These synthesized transformations can be used to propose fixes for student assignments based on corrections from other students and can also be used to automate repetitive refactoring tasks.

Raza and Gulwani [41] present an automated algorithm that attempts to predict data extraction programs only from input examples. Their tool can split system log statements into table columns or extract data from lists on webpages into spreadsheets.

In program synthesis shorter and simpler program often receive a better ranking. By also taking the execution traces of the program candidates into account, Ellis and Gulwani [42] improve the accuracy of the PROSE program learner even further. For example, a program that extracts overlapping substrings is ranked lower than a program without overlapping extractions. They also weigh the produced output of applying a program to input only examples and prefer programs which produce output structurally similar to provided output examples.

User Interaction with Programming by Example I/O examples are an ambiguous specification of a program. As such, user confidence in the correctness of the synthesized

program is important for a wide adoption of Programming by Example-based systems [43]. Miller and Myers [44] uncover outliers in input data provided to a Programming by Example text editing task. For the PROSE framework, Mayer et al. [45] compare two approaches for disambiguation by the user. With their first approach, the user can browse lower ranked program candidates in a tree-like natural language transformation of the constructed VSA. The second approach actively asks the user to resolve ambiguous output possibilities for input only examples.

Further Approaches to Regular Expression Generation from Examples Bartoli et al. [46] leverage genetic programming to generate regular expressions based on user examples. Their approach also represents the regular expressions in a tree structure and mutates them to maximize extraction accuracy on the provided examples while minimizing the length of the regular expression. Their evaluation shows that their algorithm is on average faster and more accurate than humans [47]. The WHISK system by Stephen Soderland [48] learns text extraction regular expressions for semi-structured text. The system interleaves the learning process with example annotation and reduces the number of required examples by presenting examples that eliminate ambiguities between learning candidates.

2.2.2 Information Extraction

Information extraction techniques aim at structuring unstructured information to support subsequent processing. We present existing approaches to extract information from semi-structured documents and how these approaches differ and can benefit from chunk retrieval.

The PADS project [49] is centered around declaring grammars for the information contained in semi-structured documents. Based on this declarative data description various tools are generated. This includes tools such as format converters, e.g. to XML, data adaptors to other tools, statistical analyzers and visualizers. Manually defined PADS grammars eliminate the need to develop a custom data extraction parser. Xi and Walker developed ANNE [50], which can infer context-free PADS grammars from a few user annotations and the raw document data. Fisher et al. [51] fully automated the generation of PADS grammars. They split the data into chunks, documents or lines, and further into tokens. Parentheses are used to infer hierarchical structure information. The system guesses grammar operators unifying single tokens or subexpressions, scores the resulting grammars and applies appropriate rewrite rules to refine the candidate grammars.

The IEPAD tool created by Chang et al. [52] automatically identifies data extraction patterns in semi-structured web pages, without requiring user-labeled training examples. They split the documents into tokens, discover repetitive patterns using occurrence-counting suffix trees and select the most regular and compact patterns to define extraction.

Smith and Lopez [53] extract structured information from sets of semi-structured documents that contain similar information but which are structured differently. The user provides rules for each piece of information, detailing in which section of the document it might be present. Identifying keywords and regular expressions determine whether the information is present and where it starts and ends.

These information extraction approaches support parsing and structuring of entire files. We investigate chunk retrieval techniques which do not aim at inferring the whole structure of a build log. Instead, chunk retrieval focusses on extracting one specific information characterized by the user.

2.2.3 Information Retrieval

The process of automatically selecting unstructured documents related to a given *search query* is called *information retrieval* [54]. In information retrieval, algorithms try to determine the general topic or conceptual information of a document. Usually this is done by preprocessing the documents, transforming them to a term-by-document matrix and weighing the terms with TF-IDF [55]. On the matrix the algorithms apply a similarity comparison such as, for example, vector space models to calculate the similarity of the different documents to each other [56].

Information retrieval techniques are leveraged to improve software engineering in various areas. Antoniol et al. [57] query manual pages and requirements with program identifiers to create trace links between code and documentation. The same task is addressed by Marcus et al. [58], who also incorporate source code comments into their query. Panichella et al. [56] and Runeson et al. [59] apply similar information retrieval techniques to detect duplicated bug reports. Salton et al. [60] retrieve text parts relevant for a given user query by calculating global document similarity scores and refining the output through local passage similarity. They also calculate similarity with term vectors. Our chunk retrieval technique CTS uses the same approach of term vectors to identify which lines to extract from a log file.

3 Chunk Retrieval Techniques for Build Logs

This chapter introduces the concept of chunk retrieval techniques. We use these techniques to extract text chunks from build logs that represent a specific, targeted information. This chapter first presents how a build log, which we take as input to the chunk retrieval, is created by a continuous integration (CI) build. Next, the chapter describes retrievable information chunks in build logs and gives examples found in Travis CI build logs. We illustrate why it is useful to extract the presented information chunks. The following section introduces our concept of chunk retrieval techniques. We present the three chunk retrieval techniques we investigate in this thesis: program synthesis by example (PBE), common text similarity (CTS) and keyword search (KWS).

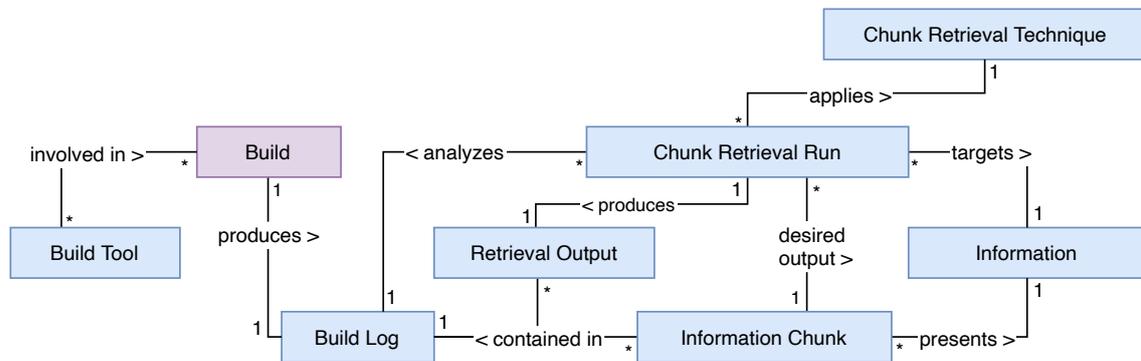


Figure 3.1: The different entities related to a CI build.

3.1 Characteristics of a Build Log

The idea of CI is to catch errors early by integrating software changes fast and often [61]. Companies link a CI server, e.g. Travis CI, to their source code repository. After making a change, the developer commits and pushes the new version of the code to the repository. A push on specific branches or the creation of a pull request triggers a CI build.

A CI build typically runs through the following stages:

- Pulling the new, changed version of the source code into the build environment.
- Building the software, i.e. compiling and packaging it [62].
- Running static analysis tools [8].
- Running automated tests [3].
- Deployment of the build artifact [63].

However, these are only *typical* stages and there is a high variability in the CI build processes of different software projects [2]. Some smaller projects might use CI to just

ensure their code compiles as a minimal check before reviewing a pull request. Other projects might have various stages of extensive automated testing.

Software tools involved in the build write out log messages to the console. They communicate progress updates, error messages and warning messages to the user [24]. We refer to the concatenation of this output as *build log*. The structure of their output is chosen by every tool themselves. Many have implicit or explicit structuring rules, some adhere to predefined standards like RSpec or PHPUnit [64, 65]. Figure 3.2 shows how different tools contribute to the whole build log.

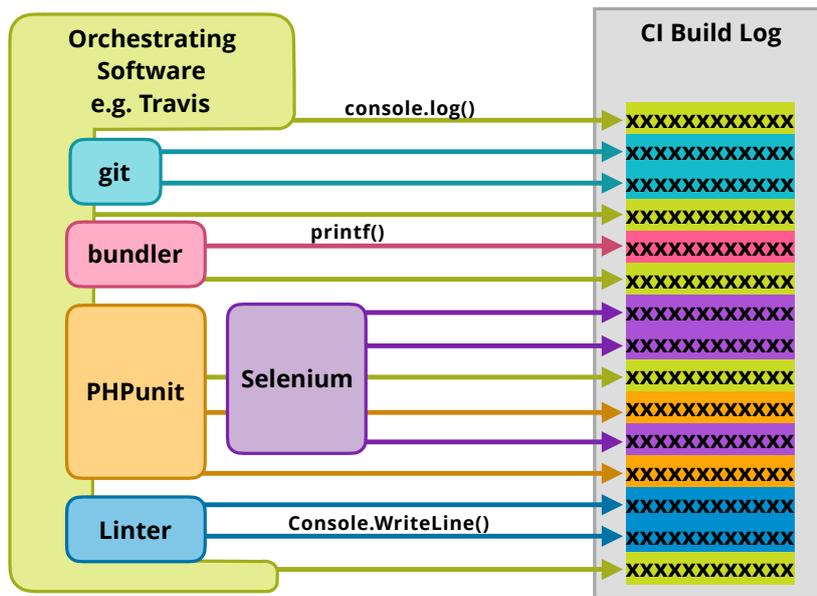


Figure 3.2: Contribution of different tools to a build log.

When analyzing build logs we do not necessarily have access to the exact build configuration, describing which tools are used in which order. We also do not necessarily have access to a useable definition of the output structure of a specific tool. Therefore, build logs are semi-structured, as described in Section 2.1.3.

3.2 Information Chunks in Build Logs

CI build logs contain a great amount of information about the CI build they correspond to. This section defines our concept of information retrievable from a CI build log. We use the introduced terms throughout the following sections and chapters to discuss about the characteristics of chunk retrieval techniques. To illustrate why chunk retrieval from build logs is useful, we present examples of information contained in Travis CI build logs and describe use cases for developers and researchers to retrieve them.

Central to this explanation is the piece of *information* possibly retrievable from a build log. *Possibly*, because an information is not necessarily present in every build log. If it is present, we call the text part which describes the information the *information chunk* or in

short *chunk*. Each chunk is always contained in a specific build log. Figure 3.1 presents the relation between an information, a chunk and other entities involved in a CI build.

Chunks can be hierarchically ordered, determined by how their textual representations contain each other. As this is an ad-hoc and a-posteriori structuring schema, as described in Section 2.1.3, this section only presents hierarchical containment for chunks whose hierarchy is known. Most chunks can appear in various hierarchical arrangements and are therefore not contained in any other chunk.

During our initial exploration and log data collection for the *LogChunks* data set, we collected a broad set of build logs from 29 languages and 87 repositories from Travis CI [18]. We inspected them to get an impression of the information one would want to retrieve from a build log. In the following, we describe examples of information chunks that can be retrieved from Travis CI build logs. All information examples containing “Travis” in their name are specific to Travis CI build logs, the others can also apply to build logs from another CI environment. Figure 3.3 shows an overview of these information examples.

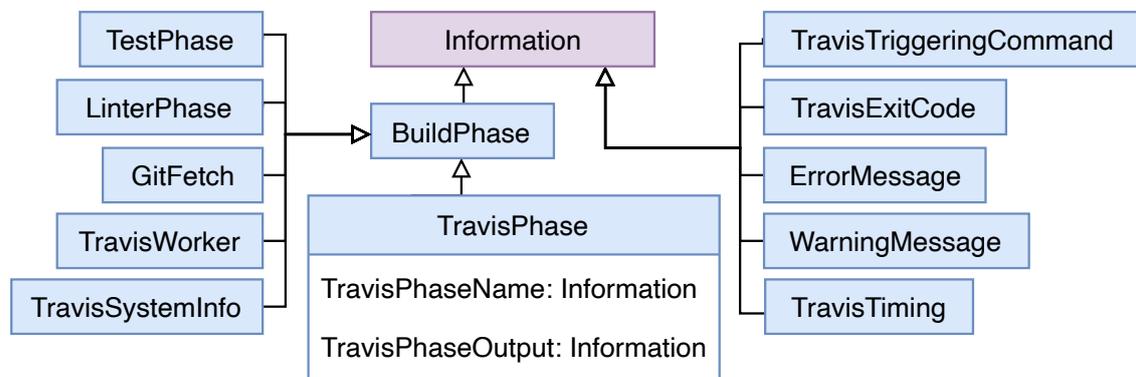


Figure 3.3: Information retrievable from build logs.

- **BuildPhase** Build logs can be divided into the sections produced by different tools. These build steps could, for example, be a **TestPhase**, **LinterPhase**, **TravisWorker** or **GitFetch**.
- **TravisPhase** Travis CI build logs consist of several build phases defined within the Travis CI configuration language. Within the build log each phase is framed by `travis_fold:start:<phase name>` and `travis_fold:end:<phase name>`.

A **TravisPhase** contains:

- **TravisPhaseName** The string Travis CI uses to identify the phase in start and end statements.
- **TravisPhaseOutput** The output generated during the **TravisPhase**. The chunk presenting the **TravisPhaseOutput** is the string between the start and end statements.

Retrieving the names of the phases of a Travis CI build log could be used to later reconstruct an overview of the executed steps within a build. Figure 3.4 shows an example of a **TravisPhase** chunk and its components within a build log.

- **TravisTiming** Travis can measure the time of specified sections of the build process. A developer can retrieve this timing information to automatically monitor the performance of their build. Figure 3.4 shows an example of a TravisTiming chunk.
- **TravisWorker** Travis CI logs which machine is executing each build. Retrieving this information from multiple build logs can help visualize the impact of the build server assignment algorithm. The TravisWorker is a good example that chunks belonging to the same information can have different textual representations in different build logs. Figure 3.5 and Figure 3.6 show examples of different TravisWorker chunks.
- **TravisSystemInfo** At the beginning of each log Travis CI describes the tech stack of the server executing the build. A developer can retrieve the system information from both failing and successful logs to identify if a failure could be based on the execution environment. Figure 3.5 shows an example of a TravisSystemInfo chunk within a build log.
- **TravisTriggeringCommand** Travis CI logs the commands it uses to call certain tools. These come from the `travis.yml` configuring the build. This information can be useful for a researcher to retrieve when they reverse engineer the build configuration. Figure 3.7 shows an example of a TravisTriggeringCommand chunk.
- **TravisExitCode** Travis CI prints all exit codes of commands. A researcher can retrieve these chunks to fill an overview of the build steps and why they failed. Figure 3.7 shows an example of a TravisExitCode chunk within a build log.
- **ErrorMessage** Various tools involved in the build process output messages of errors that occurred during their execution. A developer can retrieve them to understand why the build failed. Figure 3.7 shows an example of an ErrorMessage chunk.
- **WarningMessage** In addition to errors, tools also print warning messages. Developers can collect and count them to encourage their team to resolve them. Figure 2.1 and Figure 2.2 show two examples of WarningMessage chunks formatted differently in two build logs.

3.3 Characteristics of Chunk Retrieval Techniques

For this thesis, we want to evaluate different techniques to retrieve information chunks from build logs, which we call *chunk retrieval techniques*. The techniques we investigate do not require to parse the structure of a whole build log, but focus on extracting just one specific information.

The user provides a *configuration* that specifies which information the chunk retrieval should target and supplies the necessary information for the technique to identify the targeted information chunk in a build log. Each chunk retrieval has a specific *granularity*, i.e. the smallest retrievable text piece and uses a specific *identification technique* to select the log parts it retrieves. Each configuration addresses a specific *scope*. The scope can be a specific project, a tool involved in the build, a programming language or global, when configuring a retrieval technique for all possible build logs. A *run* of a chunk retrieval technique consumes a build log as a plain text file and produces a string output, which consists of substrings of the build log text.

```
+echo 'Pages index succesfully built.'
Pages index succesfully built.
+build_archive
+rm -f tldr.zip
+cd /home/travis/build/tldr-pages/tldr/
+zip -q -r tldr.zip pages pages.fr pages.it pages.pt-BR pages.ta pages.zh LICENSE.md index.json
+echo 'Pages archive succesfully built.'
Pages archive succesfully built.
travis_time:end:223632fa:start=1564389017210409855,finish=1564389017771715502,duration=561305647
OK [32;1mThe command "bash scripts/build.sh" exited with 0.

TravisTiming

-----
travis_fold:start:after_failure TravisPhaseName
OKtravis_time:start:15e94798
OK$ python scripts/send_to_bot.py < test_result
/home/travis/.travis/functions: line 104: test_result: No such file or directory
travis_time:end:15e94798:start=156438901777138553,finish=1564389017781000644,duration=3862091
OKtravis_fold:end:after_failure
OK
Done. Your build exited with 1.

TravisPhaseOutput
```

Figure 3.4: Excerpt from a build log showing a TravisTiming chunk and a TravisPhase chunk, containing the TravisPhaseName chunk and the TravisPhaseOutput chunk.

```
travis_fold:start:worker_info
OK [33;1mWorker information [0m
hostname: 79a22981-96ed-48a3-83d3-6e2e46269c43@1.production-2-worker-org-gce-2qgb
version: v6.2.0 https://github.com/travis-ci/worker/tree/5e5476e01646095f48eec13196fdb3faf8f5cbf7
instance: travis-job-8de07657-73ec-4573-bd82-b96e805f75b4 travis-ci-garnet-trusty-1512502259-986baf0 (via amqp)
startup: 6.547381796s
travis_fold:end:worker_info
OKtravis_fold:start:system_info
OK [33;1mBuild system information [0m
Build language: ruby
Build group: stable
Build dist: trusty
Build id: 517100249
Job id: 517100251
Runtime kernel version: 4.4.0-101-generic
travis-build version: d6b12fc73
[34m [1mBuild image provisioning date and time [0m
Tue Dec 5 19:58:13 UTC 2017
[34m [1mOperating System Details [0m
Distributor ID: Ubuntu
Description: Ubuntu 14.04.5 LTS
Release: 14.04
Codename: trusty
[34m [1mCookbooks Version [0m
7c2c6a6 https://github.com/travis-ci/travis-cookbooks/tree/7c2c6a6

TravisWorker

TravisSystemInfo
```

Figure 3.5: Excerpt from a build log showing a long TravisWorker chunk and a TravisSystemInfo chunk.

```
Using worker: worker-linux-docker-1571fced.prod.travis-ci.org:travis-linux-10
travis_fold:start:system_info
OK [33;1mBuild system information [0m
Build language: go
Build group: stable

TravisWorker
```

Figure 3.6: Excerpt from a build log showing a short TravisWorker chunk.

3 Chunk Retrieval Techniques for Build Logs

```

travis_time:end:02642577:start=1564388500814621769,finish=1564388504668035858,duration=3853414089
= [0Ktravis_fold:end:install.npm
= [0K
travis_time:start:39e44a58
= [0K$ npm test TravisTriggeringCommand

> tldr@1.0.0 test /home/travis/build/tldr-pages/tldr
> bash -c 'markdownlint pages/**/*.md && tldr-lint ./pages 2>&1 | tee test_result; test ${PIPESTATUS[0]} -eq 0'

Error: Parse error on line 12:
... as the sourcedir.**Common builders:**...
-----^
Expecting 'BACKTICK', got 'TEXT'
pages/common/sphinx-build.md:4: TLDR003 Descriptions should start with a capital letter
pages/common/sphinx-build.md:4: TLDR004 Command descriptions should end in a period
pages/common/sphinx-build.md:10: TLDR102 Example description probably not properly annotated
= [37;40m npm = [0m = [0m = [31;40m ERR! = [0m = [35m = [0m Test failed. See above for more details.
= [0mtravis_time:end:39e44a58:start=1564388504673396790,finish=1564388506709672307,duration=2036275517
= [0K= [31;1m The command "npm test" exited with 1. = [0m
TravisExitCode

travis_time:start:0a79a25e
= [0K$ bash scripts/build.sh TravisTriggeringCommand
+initialize
+'[' -z '' ']'

```

Figure 3.7: Excerpt from a build log showing an *ErrorMessage* chunk, an *ExitCode* chunk and a *TravisTriggeringCommand* chunk.

The following sections of this chapter introduce the three chunk retrieval techniques we investigate: program synthesis by example (PBE), common text similarity (CTS), and keyword search (KWS). Lastly we describe other techniques which can also be treated as chunk retrieval techniques. Table 3.1 shows a comparison of the presented techniques.

Table 3.1: Overview of the described chunk retrieval techniques.

Name	Acronym	Identification Technique	Granularity	Configuration
Program Synthesis by Example	PBE	Regular expression program	Character	In/output examples
Common Text Similarity	CTS	TF-IDF & cosine similarity, expected number of lines	Line	Output examples
Keyword Search	KWS	Keywords, expected number of lines	Line	Keywords, context length
Random Line Retrieval	RLR	Random sample	Line	Retrieval length
Diff Approach	—	Line not present in successful log, information retrieval	Line	Logs from failing and successful builds

3.3.1 Program Synthesis by Example (PBE)

Programming by Example is a technique which synthesizes programs according to in- and output examples provided by the user. It enables users to create programs without a priori programming knowledge [45]. In the context of text extraction through regular expressions, Programming by Example relieves the developer from having to understand the whole document structure to solve a single extraction task [34]. In this work, we refer to our interpretation of Programming by Example as *PBE*. We investigate the suitability of PBE to retrieve information chunks from build logs. In the following, we explain the configuration and application of PBE to chunk retrieval from CI build logs.

Configuration In/output examples are the main driver of Programming by Example. We refer to in/output examples as *examples*. When retrieving information chunks from build logs the *input* is a whole build log, i.e. the whole text of the build log file. The *output* is a substring of the log file text, representing the substring that should be retrieved by the synthesized program when given the corresponding input file. One or multiple examples, the training examples, configure a chunk retrieval with PBE: they define the substring of a build log that should be extracted. The PROSE program synthesis then tries to construct a regular expression program consistent with all training examples. A program is consistent with an example if it returns the defined output when executed on the defined input [33]. PBE reports an error back to the user if it could not synthesize a consistent program. The program synthesis builds on the FlashExtract DSL, which in turn uses the FlashMeta algorithm. Both are described in Section 2.2.1.

Application A run of PBE takes a build log file as input and applies the synthesized regular expression program. It then returns the substring of the build log matched by the program or an empty string if the program found no match.

3.3.2 Common Text Similarity (CTS)

Text Similarity approaches are used to filter unstructured textual software artifacts [57–59, 66]. One common and simple technique is the Vector Space Model [54]. We investigate when text similarity is a suitable technique to retrieve information chunks from build logs. In the following we will explain the concept of how we apply text similarity to information retrieval from CI build logs, which we refer to as *CTS*.

Configuration To configure chunk retrieval through text similarity we chose to use the same concept of examples as for PBE. The lines of the output strings of the training examples define our search query. The algorithm splits the search query into single lines and identifies tokens, in our case words. Then we build a document-term-frequency matrix over the lines from the search query and prune very often or very rarely appearing words. Next, the algorithm applies TF-IDF to the matrix, a best practice for natural language queries [55].

Application To retrieve the desired information from a build log, we parse the whole text and process it in the same way as the output of the training examples. The algorithm calculates the cosine similarity [67] to compare each line of the build log with each line of the search query. After summing up the similarities of each build log line to all search query lines, we sort the build log lines in decreasing similarity. The average number of lines in the outputs of the training examples determines how many of the most similar lines are returned as the output of the retrieval run.

3.3.3 Keyword Search (KWS)

When developers are looking for a specific piece of information within a large amount of unstructured information, a first ad-hoc approach they use is searching for related keywords. Indeed, this was one of the most common approaches we took when searching for the reason the build failed within a log while creating our *LogChunks* data set. As this is a technique readily available in many tools developers use to view build logs, we study when such a keyword search is suitable for retrieving information chunks from CI build logs. In the following we will explain how we use simple keyword search to retrieve information from CI build logs, which we refer to as *KWS*.

Configuration A set of keywords configures the chunk retrieval with KWS. To better compare KWS with PBE and CTS, we also configure it through examples. We link each example with keywords, which appear in the targeted chunk or close to it in the input build log. The configuring keywords for KWS are the ones that appear most often in the keywords of all training examples.

Application For a retrieval run, we take a whole build log file as input and search for all exact occurrences of the keywords. As keywords are often not directly describing the desired information, but rather appear close to the desired information, KWS also retrieves the lines around the found keyword. The number of surrounding lines retrieved is the average of lines in the output of the training examples.

3.3.4 Other Techniques

Log Diff Amar et al. use a technique based on line diffs and information retrieval to identify relevant lines from a failed build log [12], as we describe in more detail in Section 2.1.2. The configuration for the technique is the log from the last successful build and relevant past failures. This technique retrieves the lines from a build log that are not present in the successful build log and contain terms related to the given past failures.

Random Line Retrieval (RLR) In our evaluation, we want to compare against a baseline of randomly extracted lines. The average number of lines in the outputs of the training examples is the configuration for random line retrieval (RLR). It retrieves this number of lines randomly sampled from the build log.

3.4 Tool Implementation

For our comparison study we implemented PBE, CTS, KWS and RLR and a unifying interface. The unified interface is implemented in Ruby and calls the separate technique implementations over the command line. The implementation of PBE in C# is based on the Microsoft PROSE library [31]. We implemented CTS, KWS and RLR using R and the text2vec library [68].

4 LogChunks Data Set

This chapter describes the creation of the *LogChunks* data set that we use in our empirical comparison of chunk retrieval techniques. *LogChunks* is a collection of 797 Travis CI build logs from 80 GitHub repositories spread over 29 programming languages. For each build log we manually label which substring describes why the build failed. The data set also provides keywords we would use to search for the labeled log chunk and categorizes the log chunks according to their format within the log.

We start this chapter by explaining why we created *LogChunks* and how it enables us to compare the chunk retrieval techniques PBE, CTS and KWS. Then, we introduce related data sets and show how *LogChunks* differs. The chapter describes the data schema of *LogChunks* and our log collection process. Further, it presents the labeling process and how we validated the labeled data, including a survey of the original developers of the projects represented in *LogChunks*.

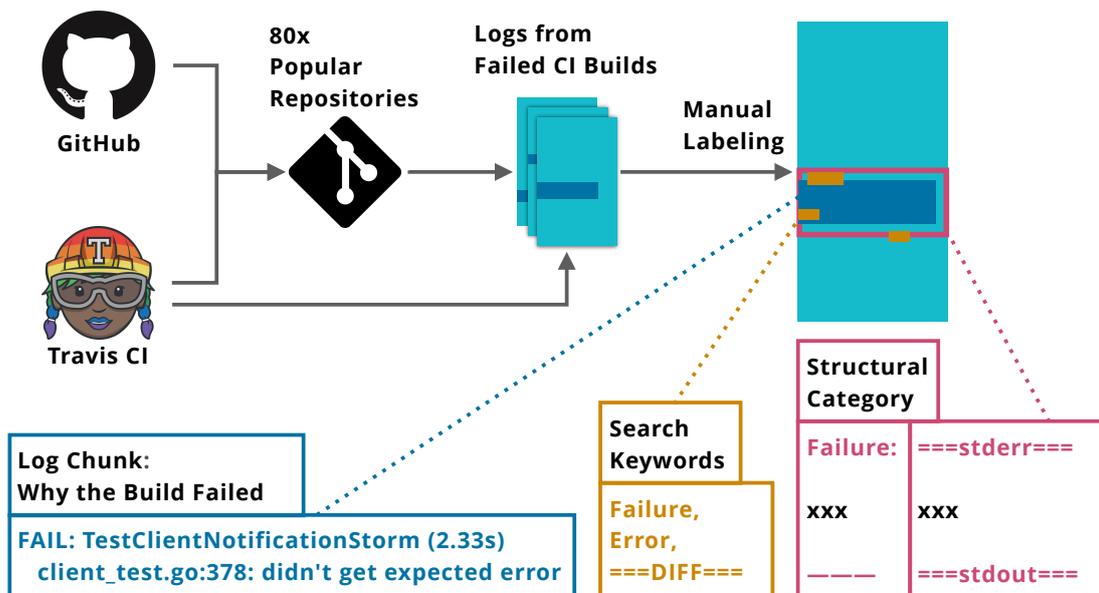


Figure 4.1: Overview of LogChunks.

4.1 Motivation

We create *LogChunks* as a data set for our empirical comparison study of the three chunk retrieval techniques PBE, CTS and KWS. In the following, we motivate which repositories and build logs we collect and which data we label for each build log.

Log Collection Existing build log analysis approaches, such as writing a custom parser and creating regular expressions, are specific to the build tools which create the targeted

log output. In contrast to that, our findings about the chunk retrieval techniques should not be specific to a particular build tool. As the main development language in a repository influences the build tools, we want our comparison study and *LogChunks* to cover a wide array of development languages. This is why we selected GitHub repositories from 29 different main development languages for our data set.

Examples configure the chunk retrieval techniques we investigate. As the structure of a build log changes from project to project, examples on logs from one project are not suited to configure a retrieval on logs from another project that uses different build tools. Therefore, a user of a chunk retrieval technique has to manually create the examples for each targeted information and project. The effort of creating these examples should not be too high, e.g. as high as developing a custom extraction program. Our study should investigate how good the chunk retrieval techniques perform with a small number of training examples. For that reason, we choose to cover a wide range of 80 repositories and include only a few, namely ten, logs and examples per repository.

Targeted Log Chunk To evaluate chunk retrieval techniques, we have to select an information chunk the techniques should target. The targeted information chunk should occur as a realistic use case for developers or researchers and should not be trivial for any of the studied techniques.

As the targeted log chunk we choose the part of the log describing why the build failed. Developers examine build logs to identify the cause of a build failure so they can address the issue. As we described in Section 2.1.1, various researchers analyze build logs for the reason the build failed.

In addition, the log chunk describing why a build failed can be represented in different ways in build logs. The representation and content of this log chunk depends on the tool reporting the fault and a build can fail in various of the tools involved in the build. This makes it non-trivial for any of the three techniques to extract. An example of a log chunk easily extractable by all of the three techniques is the *TravisTiming*, described in Section 3.2. Travis CI logs its timing statements in an always consistent structure, therefore it would be easy for PBE to synthesize a regular expression program to extract them. The same holds for search keywords identifying the lines which contain the *TravisTiming* and selecting lines with similar text content.

Additional Data: Keywords and Structural Categories For each log chunk example we additionally annotated keywords we would use to search for this log chunk, as well as a categorization of the chunks into structural categories according to their format within the log. The technique KWS is configured by keywords instead of in/output examples like the other techniques. To make the evaluation more consistent and comparable we choose to annotate search keywords to each log chunk example. We can then use these extended examples as a configuration for KWS. An increasing number of examples is then comparable to the user basing his given keywords on more previously seen build logs.

For our second research question we want to investigate how structurally similar the training examples have to be for a chunk retrieval technique to be applicable. To measure

this in our study we choose to categorize the examples according to their format within the log. Section 4.3 gives examples of log chunks from the same and from different structural categories. As we evaluate the techniques separately for separate projects, the categorization is relative to the project which produced the build log.

4.2 Related Data Sets

This section presents existing data sets of CI build logs and why *LogChunks* is unique among them.

TravisTorrent The *TravisTorrent* data set [17] collects a broad range of metadata about builds on Travis CI. It combines data accessible through the public Travis CI API [69], related data from GHTorrent [70], the corresponding git repository and data obtained through analysis of build logs. The data obtained through build logs contains the names of failing test cases, similar to the chunk describing why the build failed in *LogChunks*. However, these values are obtained through a manually developed parser, which only supports specific Ruby test runners and Java Maven or JUnit logs. *LogChunks* provides manually labeled data on the description of why the build failed for a much wider selection of programming languages.

Travis CI Build Log Data Set Lorient et al. [71, 72] collected a large amount of Travis CI build logs from 130 GitHub repositories to analyze their use of the Checkstyle plugin. They selected Maven repositories that included the Checkstyle plugin and also used Travis CI. Their data set only provides the plain build logs, whereas *LogChunks* additionally provides manually labeled data about the chunk describing why a build failed.

4.3 Data Schema

This section describes the file structure and the data schema of *LogChunks*. We give a detailed explanation for the manually labeled data.

LogChunks has two top level folders, `build-failure-reason` and `logs`. Each contains folders representing the main languages of the repositories in *LogChunks*.

The logs are organized in folders for each repository. Within each repository folder the logs are separated according to build status. Currently *LogChunks* only contains logs from failed builds. The build status folder contains the full logs in files named with the ID of the Travis CI build that produced the log.

The folder `build-failure-reason` contains the manually labeled data of *LogChunks*. The data set provides an XML file for each repository. Figure 4.2 presents the schema of these XML files.

For each repository, *LogChunks* gives about 10 Examples. Each Example consists of:

- **Log:** the relative path to the input build log.
- **Chunk:** the targeted information chunk. We are targeting the substring of the log that describes why the build failed.

```
1 <Examples>
2   <Example>
3     <Log>C/php@php-src/failed/529279089.log</Log>
4     <Keywords>ERROR, FAIL, DIFF</Keywords>
5     <Category>0</Category>
6     <Chunk>001+  ERROR: process timed out
7
8 001- OK.
9 =====DONE=====
10 FAIL Bug #60120 (proc_open hangs when data in stdin/out/
    ↳ err is getting larger or equal to 2048) [ext/
    ↳ standard/tests/file/bug60120.phpt]</Chunk>
11 </Example>
12   . . .
13 </Examples>
```

Figure 4.2: Example XML file from LogChunks.

- **Keywords:** keywords we would use to search for the log chunk.
- **Category:** a categorization of the structural representation of the log chunk within the build log. The category is relative to the other examples for the same repository.

In the following, this section defines in more detail the labeled log chunk, search keywords and structural categories.

Chunk That Describes Why The Build Failed The Chunk is the substring of the build logs that describes why the build failed. This can be the failing test case, the description of a failed linter rule or a compiler error. The Chunk is one continuous string cut from the build log. If there are multiple errors leading for the build to fail, the substring contains the first appearing continuous error descriptions. *Continuous* means that no lines reporting normal build behavior are interrupting the error descriptions. When the reason *why* the build failed was described in an external log, the Chunk includes the description *that* the build failed.

Keywords The Keywords contain a list of one to three strings appearing within the Chunk or in the area around it in the build log. We selected keywords we would use to search for the log Chunk after analyzing about 800 build logs manually. Some example keywords from *LogChunks* are: “FAIL”, “Error”, “failed”, “[−]”, “=====” or “ERR!”.

Category For each repository, we assign *structural categories* to the examples. The structural category compares how the Chunks are represented within the build logs. Build tools highlight their error messages with markings, e.g. starting each line with “ERROR”, surrounding lines filled with special characters or additional empty log lines. Two examples fall into the same structural categories if they are surrounded by similar markings. Figure 4.3 presents a log chunk from the same category as the log chunk from Listing 4.2. In comparison to that, Figure 4.4 presents a log chunk which is formatted differently within the log file. For most cases, two Chunk examples that fall into one category are outputted

```

1  =====DIFF=====
2  -----
3  005+      Parameter #1 [ <optional> $flags ]
4
5  005-      Parameter #1 [ <optional> $ar_flags ]
6  =====DONE=====
7  FAIL Bug#71412 ArrayIterator reflection parameter info [
   ↪ ext/spl/tests/bug71412.phpt]
8  -----
9  TEST 9895/13942 [2/2 concurrent test workers running]

```

Figure 4.3: Log chunk from the same structural category as the log chunk presented in Figure 4.2, “-----” separates the log chunk from one line of context.

```

1  [OK$ ./sapi/cli/php run-tests.php -P -d extension='pwd' /
   ↪ modules/zend_test.so $(if [ $ENABLE_DEBUG == 0 ];
   ↪ then ...
2  -----
3  Illegal switch 'j' specified!
4  -----
5  Synopsis:

```

Figure 4.4: Log chunk from a different structural category than the log chunk presented in Figure 4.2, “-----” separates the log chunk one line of context.

either within the same build phase or by the same build tool. For each repository, the structural categories are represented as integers, starting at 0 and increased with the next appearing category in chronological build order.

4.4 Log Collection

We describe how we select the repositories, builds and logs for *LogChunks*. To collect the build logs we built the *GHTorrentParser*, *LogCollector* and *TravisRequester* using Ruby.

Repository Sampling First, we determine a set of repositories to query logs from. Our *GHTorrentParser* queries the *GHTorrent* [70] data set for the most popular languages on GitHub [73]. It then retrieves the most popular repositories for a given language. We define *Popularity* as the number of watches. The *TravisRequester*, our tool querying the Travis API [69], can then check for a given repository whether it uses Travis CI.

For *LogChunks* we queried *GHTorrent* on 01/04/2018 for the three most popular repositories of each of the 30 most popular languages. We found 80 repositories from 29 languages that use Travis CI. Among these repositories are, for example, *git/git*, *Microsoft/TypeScript* and *jwtm/alacrity*.

Build Sampling The *LogCollector* uses the *TravisRequester* to obtain the newest builds for a given repository. It uses a stratified sampling approach: *TravisRequester*

saves the obtained builds in buckets according to their status. We encountered the following statuses during our data collection: created, started, cancelled, passed, errored and failed. The user of `TravisRequester` configures how many builds should be checked and how many builds per status should be saved.

To sample the builds for *LogChunks* we let `TravisRequester` check up to 1000 builds per repository and keep ten of the status *failed* [74]. A Travis CI build is marked as *failed* when it faults in the script section of the build configuration defined by the user.

Log Sampling For each build, the `TravisRequester` then selects a log to download. Travis CI attributes logs to *jobs*. A single build can consist of multiple jobs, e.g. building the same code version and executing tests in different testing environments. A failed build can have successful job executions, as just one failed job leads to the whole build being marked as failed. `TravisRequester` queries each build for the first job, which has the same state. For the selected jobs, the tool queries the Travis API V3 over HTTPS to obtain the corresponding build log.

We manually inspected the collected build logs and had to discard logs from three repositories. One had only a single failed build, two others had empty build logs on Travis CI. In total we collected 797 logs from 80 repositories.

4.5 Labeling Process

After collecting a wide range of Travis CI build logs we manually labeled which text chunk describes why the build failed. Following that, we 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 tried to identify the first occurrence of a description why the build failed. They copied out the first continuous description as the Chunk. They preserved whitespace and special characters, as they might be crucial to detect the targeted substring. To support exact 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.

Keywords We presented the Chunk and ten lines above and below to the labeler. Their task was to note down three strings they would put into a document search function to find this failure description. The string should appear in or around the Chunk substring and is case-sensitive. There are no special limitations on the string itself, especially spaces are also allowed.

Category To label the *structural categories* we again presented the Chunk and the surrounding context to the labeler for all logs from a repository. We asked them to assign numerical categories according to whether the Chunk had the same structural representation, i.e. the same surrounding or identifying characters. The labeler should start the categories

with 0 and increase as new ones appear. For reproducibility, we presented the logs in chronological build order.

4.6 Validation

We validate our collected data in two different ways. A different labeler performed a second pass of labeling the build failure reason, keywords and structural categories on a subset of the data. In addition, we sent out a survey to the developers, whose commits triggered the builds within our data set. We asked them whether our retrieval of the log part describing the reason the build failed was correct. This sections describes these two validation studies.

4.6.1 Inter-Rater Reliability Study

To evaluate the validity of our labeled data we performed a second labeling of a sample of the data in *LogChunks*.

Method We followed the same labeling process as described in Section 4.5. For the build failure reason and the keywords, we presented 30 randomly sampled build logs from distinct repositories in *LogChunks*. The structural categories are relative to other logs from the same repository. Therefore, we randomly sampled 3 repositories and presented all 10 examples within them to the second labeler.

Results For the substring describing why the build failed, the two labelers exactly agreed in six cases. In 15 cases the second labeler selected more lines, in five their selections overlapped and in four they completely disagreed. Regarding the keywords, the two labelers completely agreed in nine cases and completely disagreed in two cases. In nineteen cases there was overlap in the keywords of the two labelers. When classifying the chunks into structural categories, the two labelers agreed in 26 cases and disagreed in four cases.

Discussion The results of this validation study show that there is overlap in the data from both labelers, however also a high variation. We believe that the main cause for this is that our explanations to the second labeler were not extensive enough. There were implicit, inconsistent assumptions both labelers created during their work. In the following we describe these assumptions from both labelers and the implications on our description of the data classes.

For the first data class, the reason the build failed, it was ambiguous whether the labeled substring should contain the information *that* the build failed. This concerns statements like “The build exited with 1”. One labeler included such statements, while the other one only focussed on the log parts describing *why* the build failed, e.g. the name of the failing test case.

While labeling the keywords a developer would use to search for the log part describing why the build failed, the one labeler allowed arbitrary strings appearing around the presented log part. In contrast to that, the other labeler focussed on actual *words*, delimited by spaces or special characters. One labeler ignored capitalization, while the other one selected

case-sensitive keywords. A third difference was that the first labeler was presented with all substrings from a repository, yielding more general keywords than the second labeler.

For the structural categories this validation study showed a high overlap. In our instructions to the second labeler we did not emphasize the *structural* aspect enough. They sorted into categories along *why* the build failed, putting failing tests from different test runners into the same category even though the failing tests were presented differently in the log.

The main conclusion from this study is that adequately communicating all decisions and assumptions on how data is labeled is important and difficult. We reviewed the misunderstandings and incorporated more thorough descriptions of our data classes in Section 4.3.

4.6.2 Developer Survey

For *LogChunks* we analyzed around 800 build logs from different repositories and tried to extract the part of the log which describes why the respective build failed. As we were not involved in the development of any of the projects within our data set we only relied on our previous experience with various build logs and systems. We only took the logs into account and did not check the related configurations, so it is possible that we extracted parts that do describe errors but that the respective step failing is ignored by the configuration and the build failed for another reason.

The person who probably knows best why a build failed is the one committing the changes which triggered the build. If the build was e.g. part of a pull request then developer likely inspected the failed build and tried to fix the build so the pull request can be accepted. We sent out mails 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, for every build log in the data set we looked up the corresponding build and the committer information. We grouped all commits triggered by one developer and sent out an mail to each of the developers, asking whether the log chunk selected during our labeling was indeed describing the reason the build failed. Figure 4.7 shows one of the mails sent out. The mail included links to the corresponding commits, build overview and log file. We asked the developers to fill out a short survey in case our chunk was not correct. Look at Figure 4.10 to get an impression of the survey. In the survey, we presented the selected log part and asked the developer to paste in the log part actually describing the failure reason or describe in their own words why we were wrong. As some of the chunks we labeled are many lines long, we trimmed all down to 10 lines to keep the mail readable.

Results In total we sent out mails to 246 developers, asking about 3.2 build logs per mail on average. 32 of these mails could not be delivered, e.g. because they were addressed to *noreply* mail addresses. These 32 mails related to 68 of the build logs. We received answers from 61 developers, responding about 144 build logs. Figure 4.5 and Figure 4.9

show the proportions of mails delivered and logs answered about, as well as those not delivered and unanswered.

Of the 144 log chunks we received answers about, 132 were marked as describing why the build failed. 26 answered either “close, but not quite correct” or “no, the build failed for another reason”. We manually inspected the 26 negative answers and found that some stated that the proposed chunk did not show the whole description of why the build failed. This is because we had to trim long chunks to keep the mails readable. After adjusting these answers, 12 answers stated that our labeled log chunk was not correct, shown in Figure 4.8.

Discussion This study highly strengthens the trust in the validity of the extracted build failure reasons in *LogChunks*. The study received answers about 18% of the logs from *LogChunks*. After manual correction, 91% of the received answers said our labeled chunks were accurate.

One of our chunks only showed a warning and the developer proposed to also include the line above, stating that warnings are treated as errors in the build. In others that were identified as incorrect, we labeled the error message of an error that was later ignored and did not lead to the build failing.

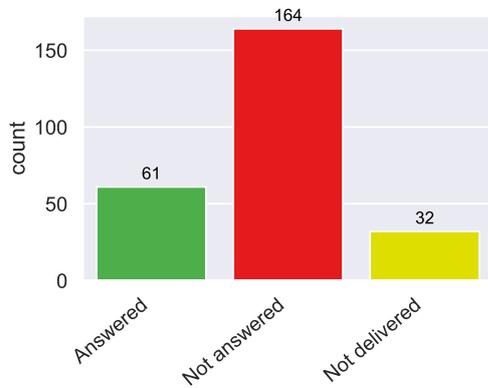


Figure 4.5: Number of mails answered, unanswered and not delivered.

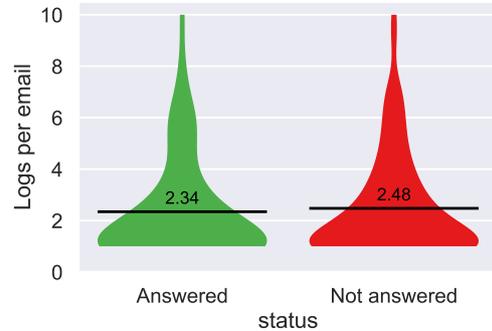


Figure 4.6: Average number of logs per mail sent out.

Hey Victor!

I am Carolin, and I am studying methods to automatically extract the reason why a build failed from a Travis CI build log.

I have seen that you are using Travis CI heavily in the dkhamsing/open-source-ios-apps repo on GitHub and came across six of your builds there. Would you help me by validating whether the reason I extracted for why the build failed is correct? Shouldn't take you more than five minutes, and helps me a lot!

1) On Tue, 23 Jul 2019 07:55:07 GMT, you authored [commit 687f378c](#), which lead to the [failed build 1630 on Travis CI](#).

I believe this is the reason why it failed (extracted from [the full Travis log](#)):

```
> Links
01. [L0191] 301 https://github.com/vanyaland/ToThePenny -> https://github.com/ivan-magda/ToThePenny
02. [L0239] 522 http://www.wtfgamersonly.com/wp-content/uploads/2014/08/gba4ios.jpg
03. [L1170] 301 https://github.com/vanyaland/Tagger -> https://github.com/ivan-magda/Tagger
04. [L1171] 301 https://github.com/vanyaland/Tagger/raw/master/Screenshots/main.png -> https://github.com/ivan-magda/Tagger/raw/master/Screenshots/main.png
05. [L1229] 301 https://github.com/vanyaland/UpcomingMovies -> https://github.com/ivan-magda/UpcomingMovies
06. [L1230] 301 https://github.com/vanyaland/UpcomingMovies/raw/master/Screenshots/movies.png -> https://github.com/ivan-magda/UpcomingMovies/raw/master/Screenshots/movies.png
07. [L1268] 301 https://github.com/vanyaland/Californication -> https://github.com/ivan-magda/Californication
08. [L1269] 301 https://github.com/vanyaland/Californication/raw/master/screenshot.png -> https://github.com/ivan-magda/Californication/raw/master/screenshot.png
09. [L1299] 301 https://github.com/vanyaland/MVVM-Example -> https://github.com/ivan-magda/MVVM-Example
```

Please help us by saying whether we were right:

[This is the correct reason!](#) (This links to a tiny thank you page to record your answer :))

[Not quite correct ...](#) (This links to a very short survey to record your answer, and if you want you can improve our work by telling us what would have been correct :D)

2) On Tue, 23 Jul 2019 07:55:07 GMT, you authored [commit d91e8725](#), which lead to the [failed build 1631 on Travis CI](#).

I believe this is the reason why it failed (extracted from [the full Travis log](#)):

Figure 4.7: An example of the mails we sent out to developers for validation of our labeled log part.

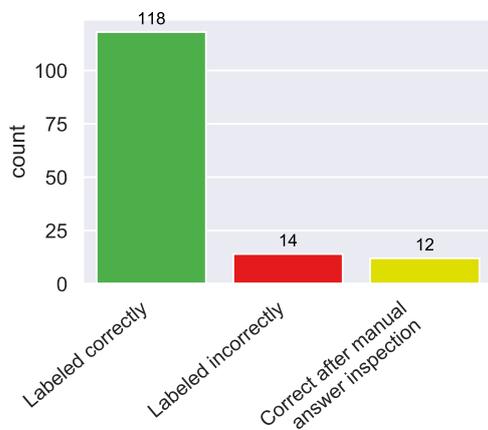


Figure 4.8: Label correctness as validated by developers.

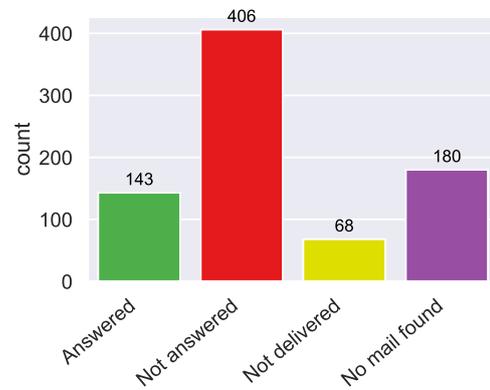


Figure 4.9: Proportions of logs about which mails were answered, unanswered, not delivered or we could not find a corresponding mail address.

Extracting the reason your build failed

Thanks a lot that you want to give us some deeper feedback!

We looked at [build 19592](#) of the repository ethereum/go-ethereum and believed that it failed for this reason:

```
FAIL: TestSimulation (10.07s)
  peer_test.go:235: failed to start the first server. err: listen udp 127.0.0.1:41777: bind: address already in use
  peer_test.go:507: Failed to start all the servers, running: 7
```

1. Was our extraction from the buildlog correct?

- yes, this part of the build log describes why the build failed
- close, but not quite correct
- no, the build failed for another reason

2. Would you go [to the whole build log](#), scroll to the part describing the actual reason the build failed and copy-paste that part back here? Alternatively you could also describe the actual failure reason with your own words.

3. If you want, please tell us why our extraction was wrong / this one is correct

Submit

Figure 4.10: Survey for the validation with developers.

5 Empirical Comparison

To investigate when PBE, CTS and KWS are suited to retrieve chunks from CI build logs we evaluate them on the *LogChunks* data set. This chapter describes our study design and which metrics we measure to answer our research questions. In the presentation of the results, we first focus on each of the three techniques and later compare them against each other.

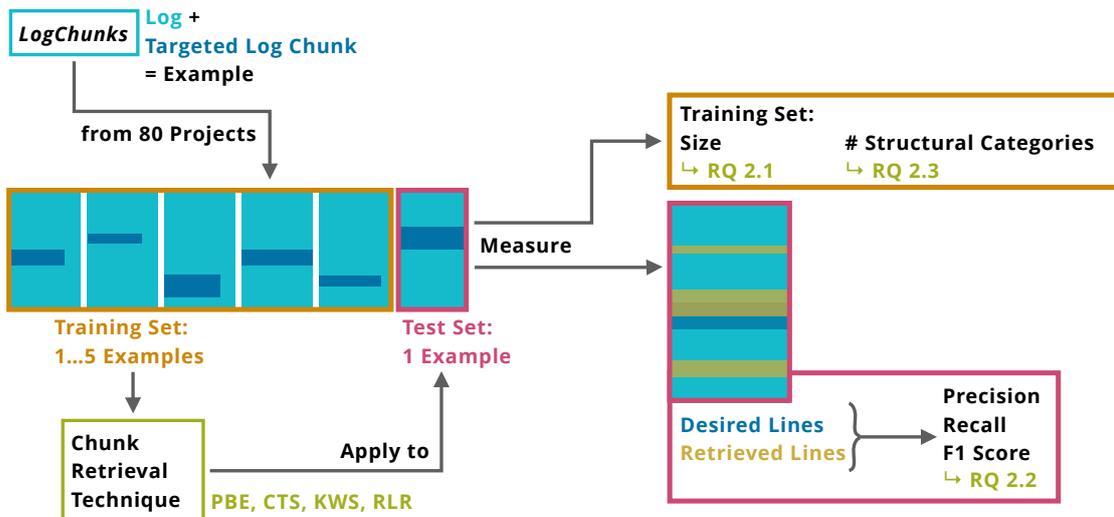


Figure 5.1: Study design of our technique comparison study.

Research Questions

- RQ1:** Which criteria influence the suitability of a chunk retrieval technique for CI build logs?
- RQ2:** Under which conditions are PBE, CTS, and KWS suited to retrieve information from CI build logs?
- RQ2.1:** How many examples do PBE, CTS, and KWS need to perform best?
- RQ2.2:** How structurally similar do the examples for PBE, CTS and KWS need to be for the techniques to be applicable?
- RQ2.3:** How accurate are the retrievals of PBE, CTS, and KWS?

5.1 Study Design

For the comparison, we evaluate the three chunk retrieval techniques PBE, CTS and KWS, described in Sections 3.3.1, 3.3.2 and 3.3.3. Random Line Retrieval (RLR), explained

in Section 3.3.4, acts as a baseline for the comparison. We run four techniques on the examples from *LogChunks*.

Training and Test Set We use *LogChunks* as the data set for our study. For each one of 80 repositories it contains about 10 build logs, manually labeled with the substring describing the reason the build failed. In addition to that, it contains keywords to search for that substring and which structural category the substring belongs to.

For each repository in *LogChunks*, we split the examples chronologically into training and test set. Therefore, we train on examples from past build logs and test on more recent build logs.

RQ 2.1: Size of Training and Test Set To analyze how many examples the chunk retrieval techniques need to perform best, we evaluate the techniques with different training set sizes. We train each technique with one to five examples from each of the repositories within *LogChunks*. The size of the test set is one.

RQ 2.2: Recording Structural Categories To determine how structurally similar the examples for the chunk retrieval techniques need to be, we record the structural categories of the examples in the training and test sets.

RQ2.3: Accuracy Metrics To measure the accuracy of the retrieved chunks we save the output lines of the chunk retrieval run on the input of the test example (*RetrievedLines*). As oracle in our evaluation, we save the desired lines from the output of the test example (*DesiredLines*).

We calculate the following metrics:

- True positives: $DesiredLines \cap RetrievedLines$
- Precision: $\frac{\#TruePositives}{\#RetrievedLines}$
- Recall: $\frac{\#TruePositives}{\#DesiredLines}$
- F₁-score: $2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$
- Successful retrieval: *true if Recall = 1*

Precision of a chunk retrieval describes which proportion of the retrieved lines were desired. Recall of a chunk retrieval describes which proportion of the targeted lines were retrieved. Throughout the presentation and discussion of our results we show these two metrics separately, as they might have a different weight when choosing a suitable technique for a task. In addition, we calculate the F₁-score, the harmonic mean of precision and recall. We define a successful retrieval as one where all desired lines were extracted, therefore when recall is one.

Recall and precision of CTS and KWS vary with the number of lines selected for retrieval. We evaluate the effect of varying the number of extracted lines by multiplying the average number of lines present in the training examples with a *retrieval size factor* from 0.5 to 2.5 in steps of 0.5.

5.2 Results

This section presents the results for PBE, CTS and KWS separately. Afterwards we compare the three techniques with each other and RLR as baseline.

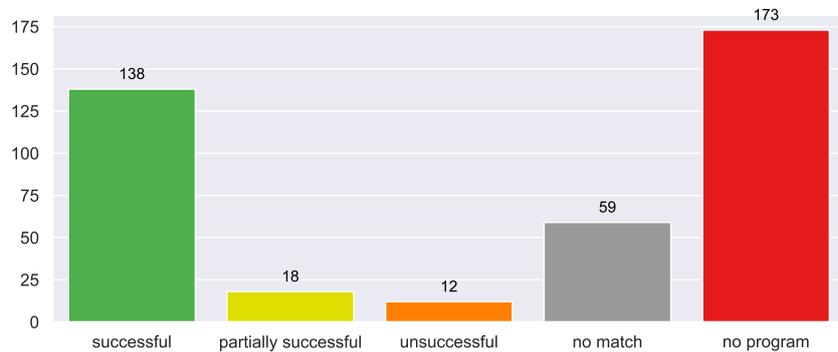


Figure 5.2: Results of chunk retrieval with PBE.

```

1 Test Output:
2 Error: Invalid CSS after "2.3em": expected expression (e.g
   ↳ . 1px, bold), was ";"
3     on line 86 of sass/components/dropdown.sass
4 Desired Test Output:
5 Error: Invalid CSS after "2.3em": expected expression (e.g
   ↳ . 1px, bold), was ";"
6     on line 86 of sass/components/dropdown.sass
7     from line 5 of sass/components/_all.sass
8     from line 6 of bulma.sass

```

Figure 5.3: Example for an unsuccessful retrieval (PBE retrieved only two of the four targeted lines).

5.2.1 Program Synthesis by Example (PBE)

Figure 5.2 shows the results of the PBE runs in our evaluation. Out of the 400 runs, 5 per each one of the 80 example sets, PBE extracted all the desired lines in 138 cases. In 89 further cases a program was also successfully synthesized, though in 59 of the 89 cases the synthesized program yielded no output at all. In 30 of the 89 cases the synthesized program did not extract all of the desired lines. For these 30 cases, the average recall was 28%. We present an example of such an *unsuccessful retrieval* in Listing 5.3, where the synthesized regular expression program only retrieved two of the four targeted lines. In 173

of the 400 cases the PROSE program synthesis could not synthesize a regular expression program that is consistent with all of the training examples.

Figure 5.4 shows the results of PBE runs compared to the number of structural categories present in the training and test examples. It shows that the program synthesis is more likely to succeed when there are only a few categories present in the training examples. When two structural categories are present, PROSE could in most cases not synthesize a program consistent with all training examples. For three or more present categories PROSE could never synthesize a consistent program.

Figure 5.5 shows precision and recall of the 227 runs where PBE could synthesize a program consistent with all training examples. When the training set size increases from one to two, recall and F_1 -score increase by about 25%, precision increases by about 10%. For two or more training examples, recall and F_1 -score stay around 75% and precision around 96%.

5.2.2 Common Text Similarity (CTS)

Figure 5.6 presents precision, recall and F_1 -score of chunk retrieval using CTS for an increasing number of training examples. When using one to five training examples, the size of the training set has no noticeable influence on precision, recall or F_1 -score of the chunk retrieval with CTS.

Figure 5.7 shows the same measurements for an increasing number of structural categories in the training and test examples. With increasing category count, precision, recall and F_1 -score decrease. Especially for more than three categories present we have no chunk retrieval runs where all desired lines were extracted.

Figure 5.8 shows the effect of the retrieval size factor on precision, recall and F_1 -score of chunk retrieval runs with CTS. The precision ranges from 52% when retrieving half expected number of lines to 25% when 2.5 times the expected number of lines. The recall ranges from 30% to 55%. A retrieval size factor of 1 gives the best average F_1 -score with 51%.

5.2.3 Keyword Search (KWS)

Figure 5.9 presents precision, recall and F_1 -score of chunk retrieval using KWS for different numbers of training examples. The recall increases by about 12% when increasing the size of the training set to more than one example, while the precision stays constant at around 16%. The F_1 -score stays around 26%.

Figure 5.10 shows the same measurements for an increasing number of structural categories in the training and test examples. For more than one structural category in the training and test examples the recall decreases by about 20% and the precision decreases about 6%. For more than two structural categories no clear trend is visible in precision, recall or F_1 -score for an increasing amount of categories in the training and test examples.

Figure 5.11 shows the effect of the retrieval size factor on precision, recall and F_1 -score of chunk retrieval runs with KWS. The precision is 19% when retrieving half of the expected number of lines. On average 9% of the lines in the build log are retrieved then. When 2.5

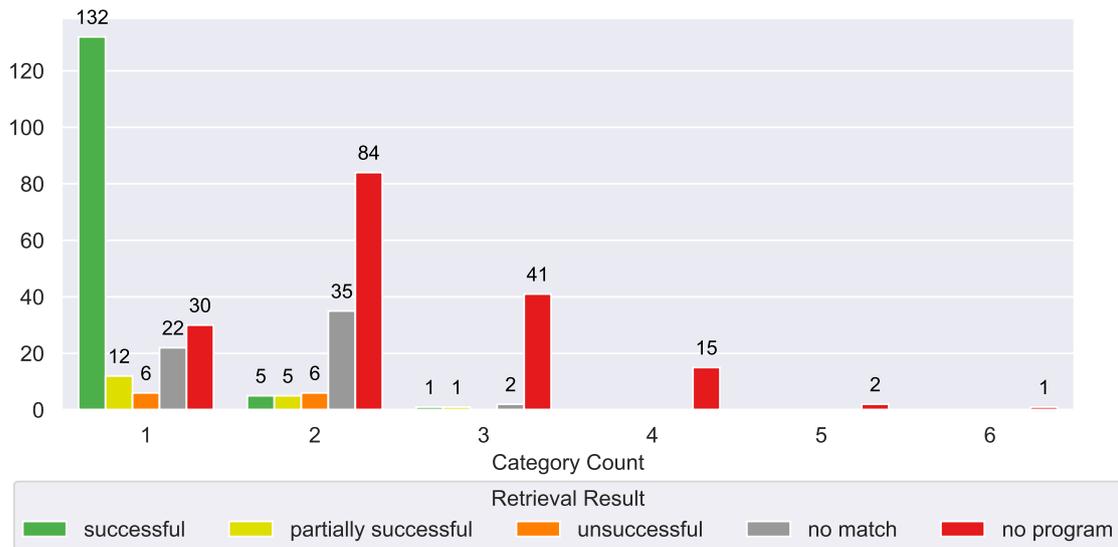


Figure 5.4: Results of chunk retrieval with PBE for an increasing number of structural categories in the training and test sets.

times the expected number of lines are retrieved, the precision decreases to 10% and a quarter of the lines in the build log are retrieved on average. The recall ranges from 58% to 75% and the F_1 -score shows a constant decrease from 29% to 17%.

5.2.4 Comparison of All Techniques

Figure 5.12 compares the success of all chunk retrieval by the different techniques in our study. CTS and KWS extract some of the desired lines in 79% and 88.5% of the chunk retrieval runs. With 38.25%, KWS also has the highest number of fully successful extractions, followed by PBE with 34.5%. PBE has the lowest number of partial retrievals with only 18 out of 400 chunk retrieval runs.

The averaged precision, recall and F_1 -score of all techniques is compared in Figure 5.13. The recall of PBE has a high skew towards one and zero, meaning in most cases either the retrieval is successful or no relevant lines are extracted at all. PBE has the highest average precision with 95%. Chunk retrieval with CTS has the highest average F_1 -score with 51% and the second highest recall with 46%. KWS has the smallest precision of the three chunk retrieval techniques. With 16% it is still higher than the precision of the RLR baseline with 7%. KWS has the highest recall of all techniques with 70%.

Figure 5.14 and Figure 5.15 show the influence of a single structural category present in the training examples compared to multiple categories present. For more than one present category, the recall of PBE decreases greatly. For CTS and KWS the values also decrease, while RLR is not affected by the number of structural categories present.

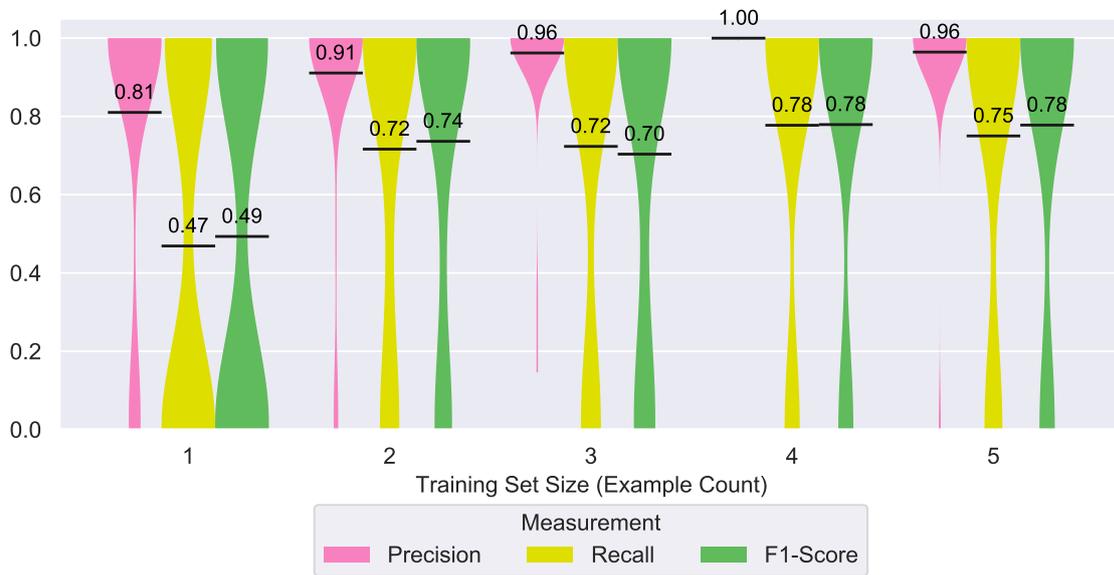


Figure 5.5: Precision, recall and F_1 -score of chunk retrieval when PBE could synthesize a consistent program compared with the size of the training set.

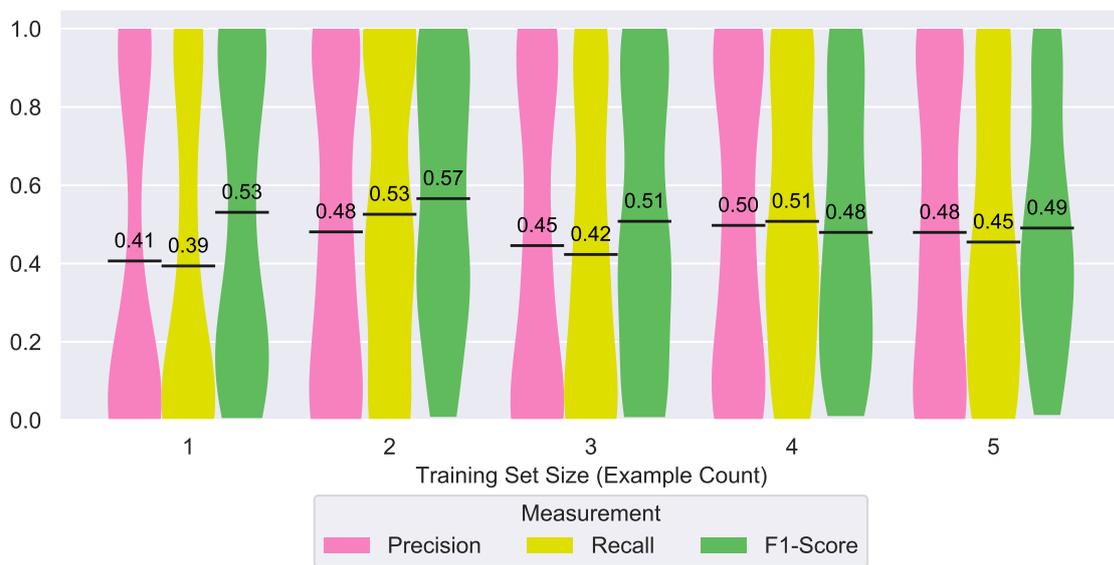


Figure 5.6: Precision, recall and F_1 -score of chunk retrieval with CTS for an increasing training set size.

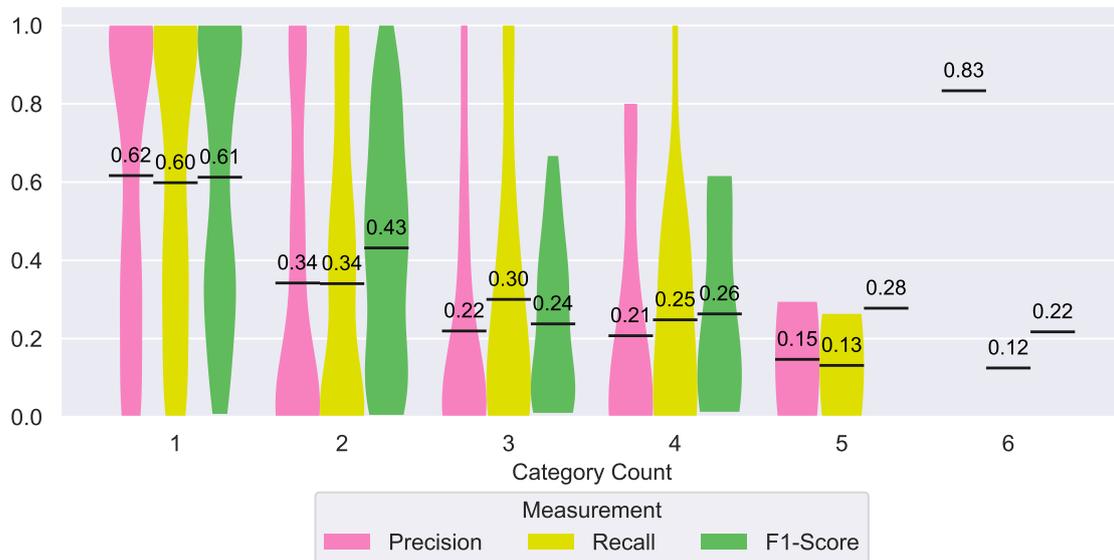


Figure 5.7: Precision, recall and F_1 -score of chunk retrieval with CTS for an increasing number of structural categories in the training and test sets.

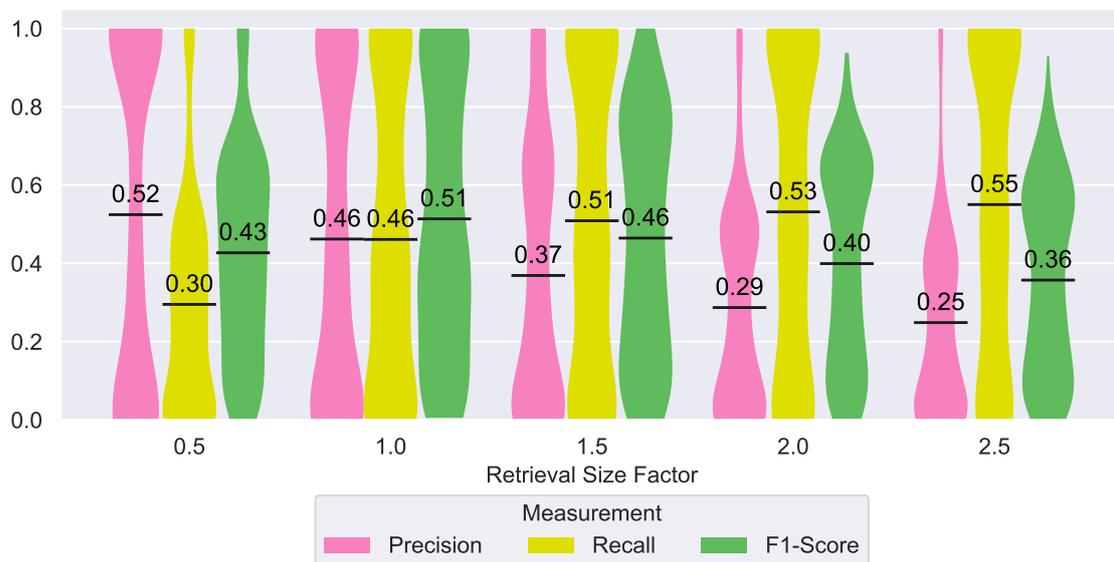


Figure 5.8: Precision, recall and F_1 -score of chunk retrieval with CTS compared to retrieval size factors.

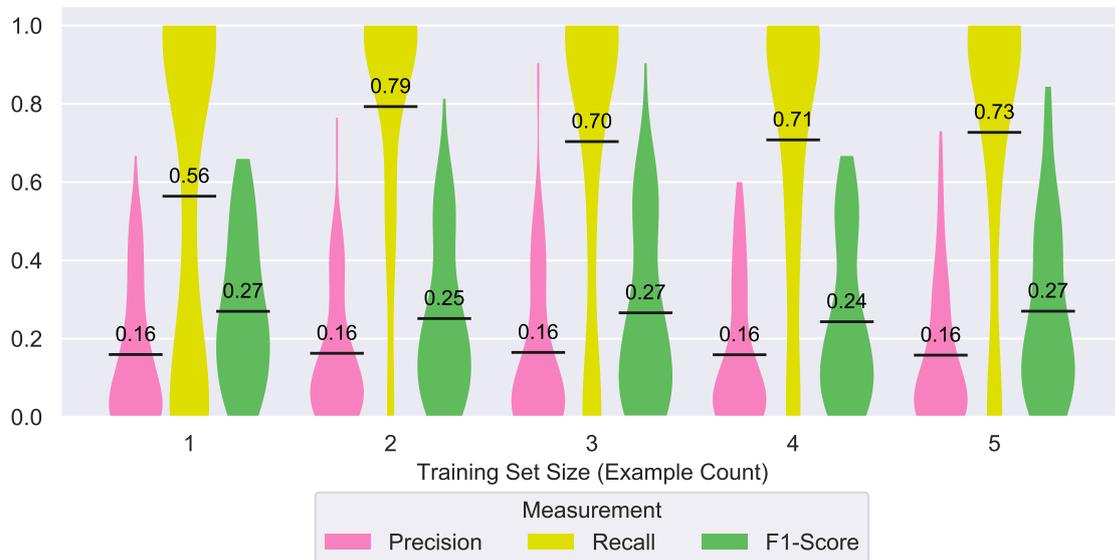


Figure 5.9: Precision, recall and F_1 -score of chunk retrieval with KWS for an increasing training set size.

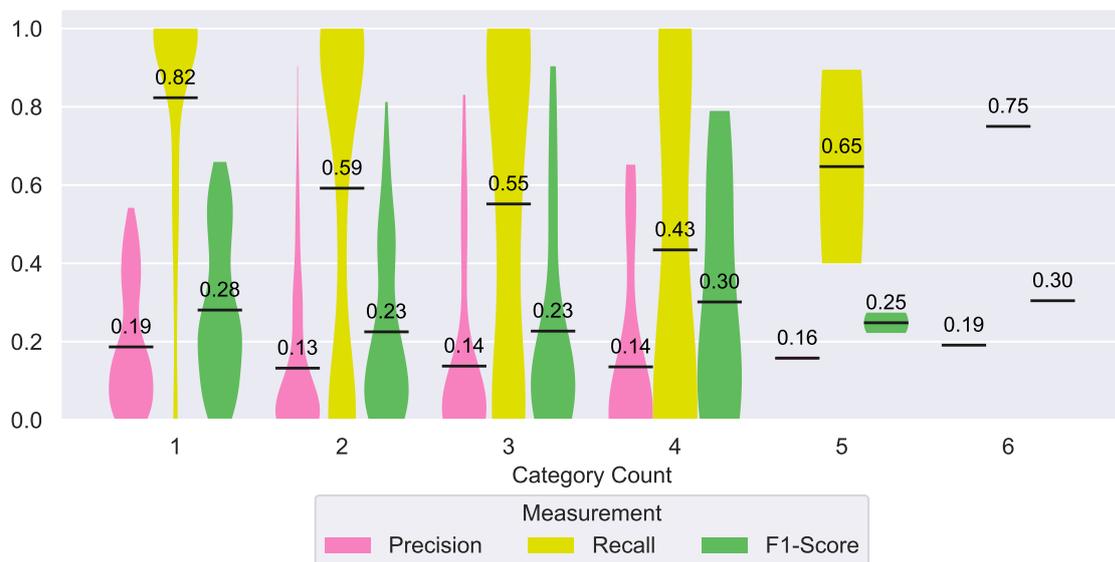


Figure 5.10: Precision, recall and F_1 -score of chunk retrieval with KWS for an increasing number of structural categories in the training and test sets.

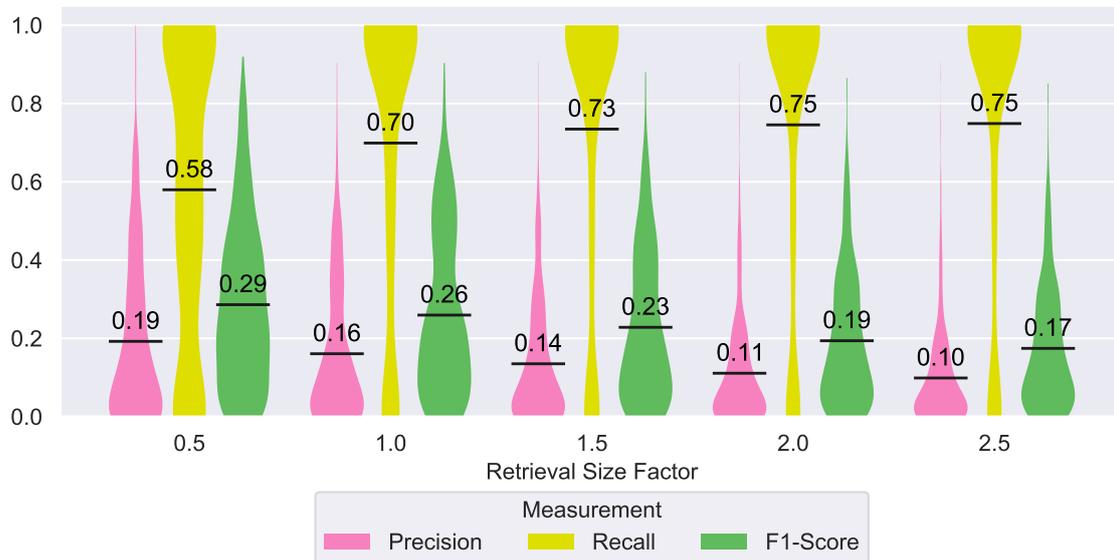


Figure 5.11: Precision, recall and F_1 -score of chunk retrieval with KWS compared to retrieval size factor.

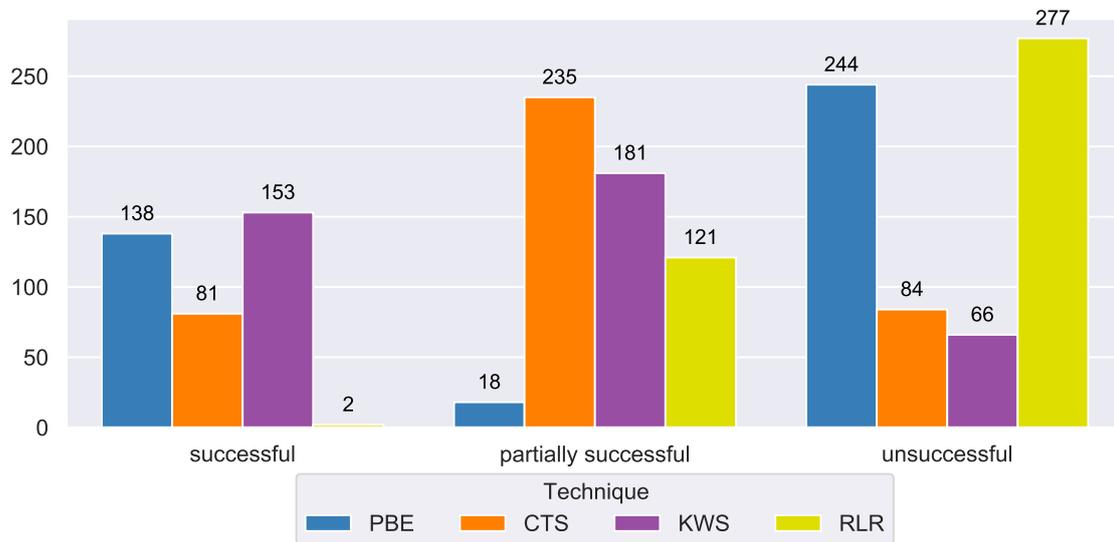


Figure 5.12: Success of chunk retrievals for all techniques.

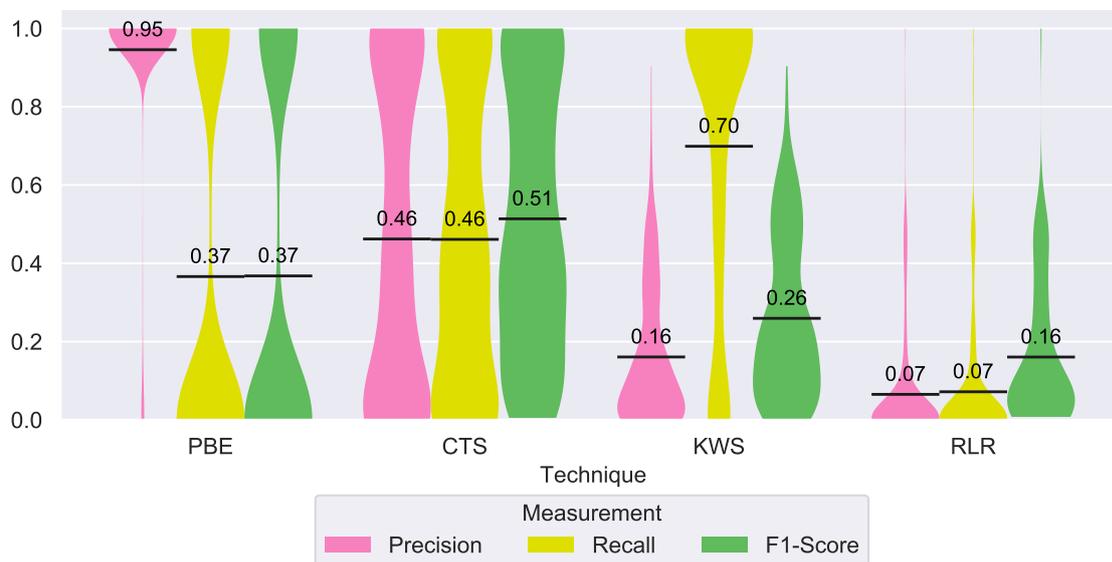


Figure 5.13: Precision, recall and F_1 -score of all techniques compared.

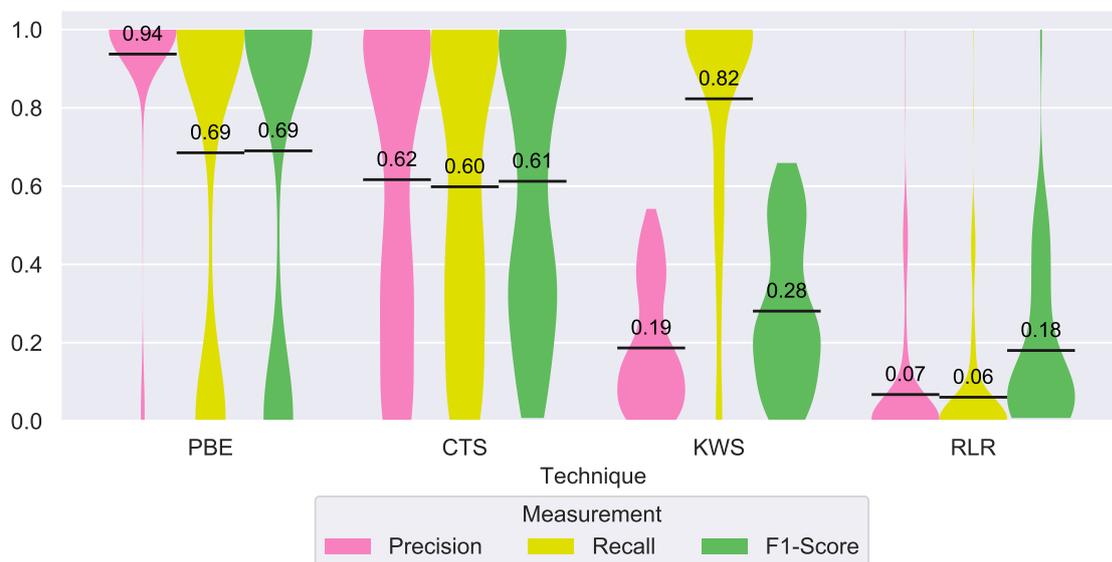


Figure 5.14: Precision, recall and F_1 -score of all techniques compared when training examples are in one structural category.

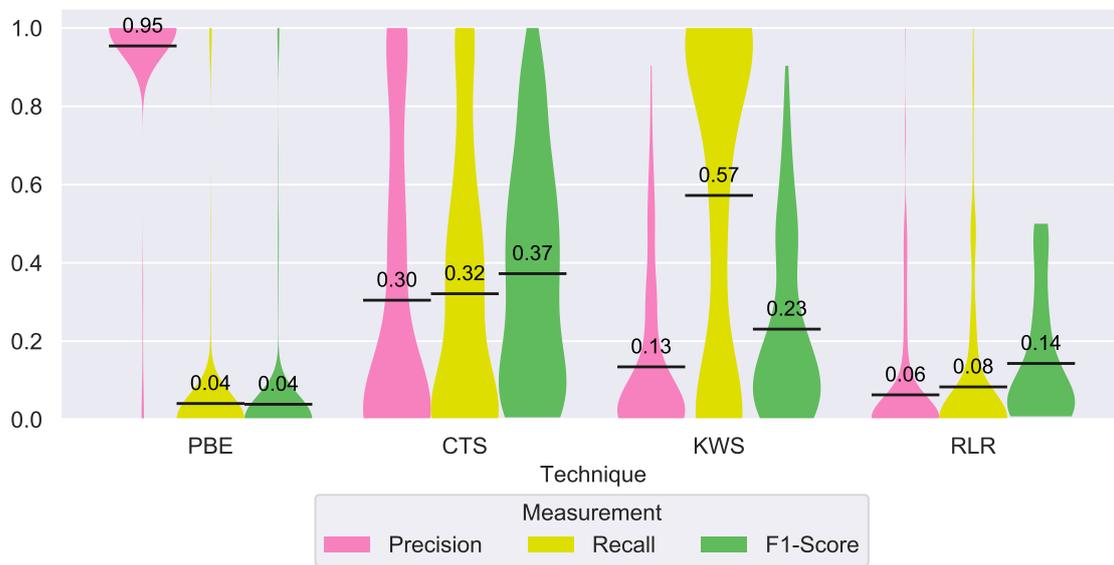


Figure 5.15: Precision, recall and F_1 -score of all techniques compared when training examples are in more than one structural categories.

6 Discussion

This chapter presents the answers to our research questions:

Research Questions

- RQ1:** Which criteria influence the suitability of a chunk retrieval technique for CI build logs?
- RQ2:** Under which conditions are PBE, CTS, and KWS suited to retrieve information from CI build logs?
- RQ2.1:** How many examples do PBE, CTS, and KWS need to perform best?
- RQ2.2:** How structurally similar do the examples for PBE, CTS and KWS need to be for the techniques to be applicable?
- RQ2.3:** How accurate are the retrievals of PBE, CTS, and KWS?

The first section discusses for PBE, CTS and KWS separately in which cases they perform best. It details for which types of input build logs, available training examples and consumption of the retrieved output each technique is suited. In the following section we discuss which of these criteria should influence the decision to use a certain technique most. Based on our empirical comparison, we present a decision tree between the three techniques we investigated.

6.1 Interpretation of Study Results

This section discusses the study results for each of the analyzed chunk retrieval techniques separately. It gives recommendations which kind of information chunk targets are best for each technique and for what kind of usage the respective output is suitable.

	PBE	CTS	KWS
Structural Categories	1	less is better	best 1 multiple okay
Training Set Size	2	no influence	2
Precision	high (if synthesis succeeds)	medium	low
Recall	high (if synthesis succeeds)	medium	high
Confidence in Output Correctness	high	low	low (precision) high (recall)
Output Consumption by	program	human	human

Table 6.1: Recommendations for each of the investigated chunk retrieval techniques.

6.1.1 Program Synthesis by Example (PBE)

Configuration and Input Our study results show that chunk retrieval with PBE gives best results when the training examples are from one structural category. This means it is suited to retrieve information chunks that always have the same surrounding structure. To extract for example the reason a build failed, the log passage describing the failure would always have to be started and ended the same way.

When the training examples are of the same structure, two examples are enough input for PROSE to synthesize a regular expression program with good recall. In our study, additional training examples did not improve the chunk retrieval.

Retrieval Output Usage If the program synthesis succeeds and applying the regular expression program yields an output, PBE shows a high precision and a high recall. The tool clearly identifies a failing program synthesis or when no output from the program applied to a build log is obtained. Therefore, if there is an output, the user can have high confidence that it is the correct output. This makes output from PBE chunk retrieval well suited for consumption by other software components.

6.1.2 Common Text Similarity (CTS)

Configuration and Input Chunk retrieval using CTS also yields better results the fewer structural categories are present in the training and test examples.

The number of training examples had no noticeable influence on precision or recall in our study. Information retrieval techniques like text similarity commonly learn on a higher number of examples than used for our study. Future work is needed to investigate how many examples yield improvements in the chunk retrieval over a single training example.

Extracting the average number of lines present in the training examples gives the best retrieval results for CTS, according to the F_1 -score.

Retrieval Output Usage CTS has good precision and recall on average, though the precision or recall of a chunk retrieval run is very hard to predict from the given result. Therefore, retrieval output from CTS is suited to be read by a human.

6.1.3 Keyword Search (KWS)

Configuration and Input KWS has a higher recall than the two other techniques for multiple structural categories present in the training and test examples. This makes KWS a good technique if there is little prior knowledge on how the targeted log chunk is represented in the build log to be analyzed. For the example of extracting the reason the build failed, KWS is best suited if a build can fail in various steps logged by different tools and no pre-categorization of where the build failed is available.

With two training examples KWS achieves good recall.

Retrieving the average number of lines present in the outputs of the training examples around every found keyword yields reasonable recall. Selecting 1.5 times as many lines

around every found keyword does improve the recall within our study but also increases the proportion of lines retrieved overall and therefore decreases precision.

Retrieval Output Usage Even though KWS has the highest recall of all three techniques, its precision is also the lowest. The output of a chunk retrieval with KWS is well suited to be read by humans.

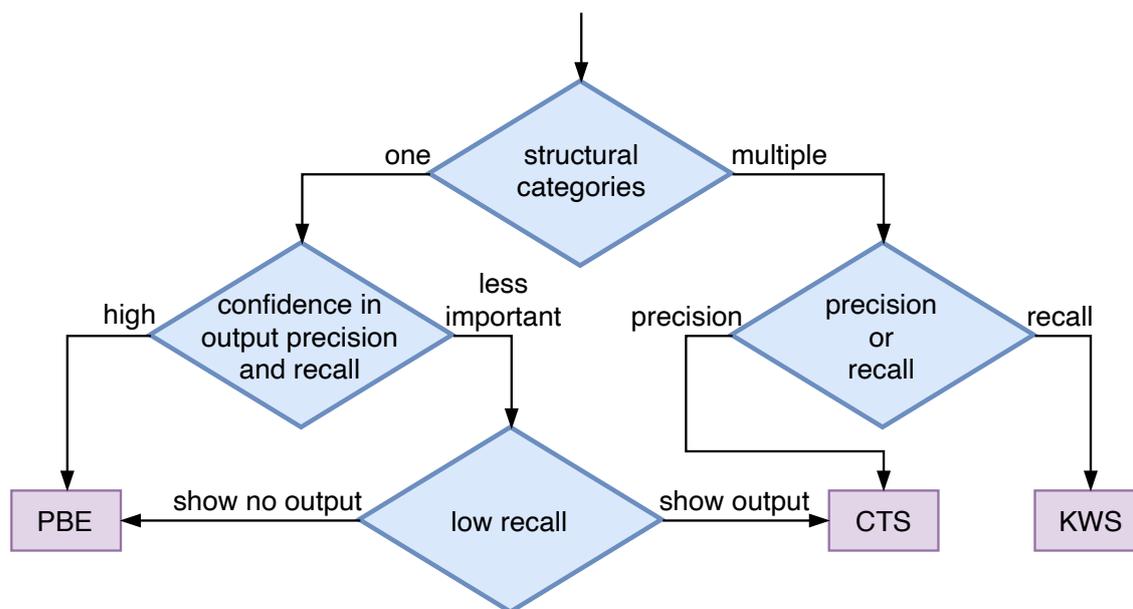


Figure 6.1: Our preliminary recommendation scheme for chunk retrieval techniques.

6.2 Recommendations of Suitable Techniques

After discussing the three chunk retrieval techniques separately we now want to unify our results into one recommendation scheme. We present a decision tree, which developers and researchers who want to retrieve information chunks from build logs can follow. The decision tree is built up of questions which either lead to more questions or to a leaf node containing a recommended technique. Figure 6.1 presents the decision tree.

Caveat! This is a preliminary theory based on the results from our comparison study. The recommendations are therefore based on our implementation of the chunk retrieval techniques as well as the logs in the *LogChunks* data set.

This decision tree is the answer to our first research question about which criteria influence the suitability of a chunk retrieval technique. The earlier in the decision tree a criterion is noted, the more important it is when distinguishing the techniques.

The first and most important aspect are the structural categories. Are the information chunks you would like to retrieve always presented in the same structural way within the build logs? Then the information chunks in all training examples and the analyzed build log are in the same structural category.

If the information chunks are from multiple structural categories, i.e. they are not represented in the same structural way within the build logs, and recall is more important than precision we recommend to use KWS. If the representations are from multiple structural categories and precision is more important than recall to the user we recommend CTS. We also recommend CTS when the representations are from one structural category, when the user does not require a high confidence in the precision or recall of the outcome and when the user would rather have output with low recall instead of no output at all. When the representations are from one structural category and the user wishes a high confidence in the correctness of the output or prefers no output over output with low recall, we recommend PBE.

Example of Using the Recommendation Scheme To illustrate how one would use our decision tree to find a suitable chunk retrieval technique we describe two concrete examples: a researcher investigating why CI builds fail and a software team wanting to monitor their build performance.

In our first example, a researcher studies whether test failures in CI are caused by a small or by a large group of test cases. They gather CI build logs from various projects, which are their only available data source. The task of the researcher is to extract the names of the failing test cases from each build log. When they use our recommendation scheme to select a chunk retrieval technique, they first have to estimate how uniform the representation of the failing test cases is in the investigated build logs. As the researcher is covering a wide range of build tools and development languages, the log chunks they target are in various, non-predictable structural representations. The next question is whether they value precision over recall. As they have to manually inspect the results of both CTS and KWS, they choose recall over precision to avoid having to inspect the whole log in case the relevant information chunk was not retrieved. Therefore, our decision tree recommends them to use KWS.

In case the researcher wants to avoid manually inspecting the retrieval results, they have to first separate the CI build logs according to the test tool responsible for logging the test results. Then the targeted log chunks are from one structural category and they can use PBE, trained with examples from each test tool separately.

In our second example, a software development team wants to monitor the performance of the phases within their CI build. They are using Travis CI, which measures the duration of build phases and documents them within the build log. As all log statements that report timing measurements are formatted the same way, the targeted log chunks are from one structural category. Therefore the development team can use PBE to retrieve the duration of a build phase as well as its name.

6.3 Threats to Validity

There are several threats to the validity of the conclusions of our work.

Implementation Our results are highly dependent on our implementation of the investigated chunk retrieval techniques and the libraries we used. Our implementation of PBE is severely based on the program synthesis provided by PROSE. Its limitations are therefore also mainly influenced by this library. For example, the need for examples from a single structural category stems from the fact that FlashExtract cannot learn regular expression programs with arbitrary boolean conjunctions and disjunctions [45]. This limitation was necessary to keep the synthesis performance reasonable.

Our implementation of CTS is dependent on the library `tex2vec` and the way they split strings into word tokens. We intentionally chose a simple, minimally configured and tuned approach to compare against. Tuning the text similarity meta-parameters more to the specific use case of chunk retrieval from build logs would yield better chunk retrieval results.

Data Set The outcomes of our comparison study are highly dependent on the build logs from the *LogChunks* data set. It only consists of build logs from open source projects and therefore it is not clear whether our results are generalizable to industry projects. We only collected build logs from Travis CI, however we chose to evaluate on an information chunk whose format is not dependent on Travis CI. This is because the reason the build failed is described within the build logs by the tools themselves and not the Travis CI environment.

Training Set Size Especially the results for CTS might be influenced by the fact that we only trained on one to five examples. We chose this small training set size as the training examples have to be provided per repository and we expect a developer to not want to provide more examples than the small numbers we evaluated on.

Few Samples with Many Structural Categories Our comparison study shows fewer measurements with many structural categories than with one category. This stems from the fact that we use a realistic data set which in many cases has only one or few structural categories.

7 Conclusion and Future Work

The goal of this thesis was to support researchers and developers in their decision on how to analyze build logs. We implemented and compared three different chunk retrieval techniques on our data set *LogChunks*, composed of 797 manually labeled build logs from a broad range of 80 repositories. Our results show that the structural representation of the targeted information in the build logs is the main factor to consider when choosing a suitable technique. Secondary factors are the desired confidence into recall and precision of the produced output and whether precision or recall is more important for the task at hand.

There are various future research opportunities based on our work:

- **Further Analysis of *LogChunks*** We created the *LogChunks* data set specifically for our comparative study, though it can be the basis for various further analyses of build log data. The keywords, for example, can be investigated to answer which keywords are used to search for the reason the build failed within build logs.
- **Cross-Repository Build Log Analysis** We trained and tested each chunk retrieval technique on examples from the same repository. We propose to analyze how techniques could be trained across repositories, building the cornerstones for build environment-agnostic analysis tools.
- **Comparison with more Chunk Retrieval Techniques** This thesis investigates the three chunk retrieval techniques PBE, CTS and KWS. Our study design can be reused to evaluate other build log analysis techniques, such as the diff and information retrieval approach by Amar et al. [12].
- **Refinement of Retrieval Quality for each Technique** We investigated basic configurations of existing techniques applied to chunk retrieval from build logs. In a next step, each of these techniques could be refined to better approach the domain of build logs. The *LogChunks* data set and our study results act as a baseline to benchmark such technique improvements. We propose the following improvements:
 - **Custom Ranking and Tokens for PBE** The program synthesis through PROSE ranks possible programs according to what the user most likely intended. One could adapt the ranking rules provided by the FlashExtract DSL to fit common build log chunk retrieval tasks. FlashExtract includes special tokens when enumerating possible regular expressions. One could extend these with tokens found in build logs, such as “-”, “=”, “ERROR” or “[OK”.
 - **Meta-Parameter Optimization for CTS** Information retrieval techniques have various meta-parameters which can be optimized for the specific use case [56]. We propose to further investigate improvements in preprocessing of the log text, in tokenization of the log lines into terms and in stop words lists.
- **Usability Analysis of Chunk Retrieval Output** Our analysis of the output produced by the chunk retrieval focuses on precision and recall. We propose to investigate how useful these outputs are to developers through controlled experiments.

List of Figures

2.1	Excerpt from a build log showing a <code>WarningMessage</code> chunk.	8
2.2	Excerpt from a build log showing a differently formatted <code>WarningMessage</code> chunk.	8
2.3	System Log Statements. Example adapted from [25].	9
2.4	Text extraction program synthesized by <code>FlashExtract</code>	11
3.1	The different entities related to a CI build.	15
3.2	Contribution of different tools to a build log.	16
3.3	Information retrievable from build logs.	17
3.4	Excerpt from a build log showing a <code>TravisTiming</code> chunk and a <code>TravisPhase</code> chunk, containing the <code>TravisPhaseName</code> chunk and the <code>TravisPhaseOutput</code> chunk.	19
3.5	Excerpt from a build log showing a long <code>TravisWorker</code> chunk and a <code>TravisSystemInfo</code> chunk.	19
3.6	Excerpt from a build log showing a short <code>TravisWorker</code> chunk.	19
3.7	Excerpt from a build log showing an <code>ErrorMessage</code> chunk, an <code>ExitCode</code> chunk and a <code>TravisTriggeringCommand</code> chunk.	20
4.1	Overview of <i>LogChunks</i>	25
4.2	Example XML file from <i>LogChunks</i>	28
4.3	Log chunk from the <i>same</i> structural category as the log chunk presented in Figure 4.2.	29
4.4	Log chunk from a <i>different</i> structural category than the log chunk presented in Figure 4.2.	29
4.5	Number of mails answered, unanswered and not delivered.	34
4.6	Average number of logs per mail sent out.	34
4.7	An example of the mails we sent out to developers for validation of our labeled log part.	34
4.8	Label correctness as validated by developers.	35
4.9	Proportions of logs about which mails were answered, unanswered, not delivered or we could not find a corresponding mail address.	35
4.10	Survey for the validation with developers.	35
5.1	Study design of our technique comparison study.	37
5.2	Results of chunk retrieval with PBE.	39
5.3	Example for an unsuccessful retrieval (PBE retrieved only two of the four targeted lines).	39
5.4	Results of chunk retrieval with PBE for an increasing number of structural categories in the training and test sets.	41
5.5	Precision, recall and F_1 -score of chunk retrieval when PBE could synthesize a consistent program compared with the size of the training set.	42

5.6	Precision, recall and F_1 -score of chunk retrieval with CTS for an increasing training set size.	42
5.7	Precision, recall and F_1 -score of chunk retrieval with CTS for an increasing number of structural categories in the training and test sets.	43
5.8	Precision, recall and F_1 -score of chunk retrieval with CTS compared to retrieval size factors.	43
5.9	Precision, recall and F_1 -score of chunk retrieval with KWS for an increasing training set size.	44
5.10	Precision, recall and F_1 -score of chunk retrieval with KWS for an increasing number of structural categories in the training and test sets.	44
5.11	Precision, recall and F_1 -score of chunk retrieval with KWS compared to retrieval size factor.	45
5.12	Success of chunk retrievals for all techniques.	45
5.13	Precision, recall and F_1 -score of all techniques compared.	46
5.14	Precision, recall and F_1 -score of all techniques compared when training examples are in <i>one</i> structural category.	46
5.15	Precision, recall and F_1 -score of all techniques compared when training examples are in <i>more than one</i> structural categories.	47
6.1	Our preliminary recommendation scheme for chunk retrieval techniques. . .	51

List of Tables

3.1	Overview of the described chunk retrieval techniques.	20
6.1	Recommendations for each of the investigated chunk retrieval techniques. .	49

Bibliography

- [1] Michael Hilton et al. “Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects”. In: *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*. ACM. 2016, pp. 426–437.
- [2] Daniel Ståhl and Jan Bosch. “Modeling Continuous Integration Practice Differences in Industry Software Development”. In: *Journal of Systems and Software* 87 (2014), pp. 48–59.
- [3] Moritz Beller, Georgios Gousios, and Andy Zaidman. “Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub”. In: *2017 14th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE. 2017, pp. 356–367.
- [4] Bogdan Vasilescu et al. “Quality and productivity outcomes relating to continuous integration in GitHubQuality and Productivity Outcomes Relating to Continuous Integration in GitHub”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 805–816.
- [5] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education, 2007.
- [6] Ade Miller. “A Hundred Days of Continuous Integration”. In: *Agile 2008 Conference*. IEEE. 2008, pp. 289–293.
- [7] John Downs, Beryl Plimmer, and John G Hosking. “Ambient Awareness of Build Status in Collocated Software Teams”. In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 507–517.
- [8] Fiorella Zampetti et al. “How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines”. In: *2017 14th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE. 2017, pp. 334–344.
- [9] Hyunmin Seo et al. “Programmers’ Build Errors: A Case Study (At Google)”. In: *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM. 2014, pp. 724–734.
- [10] Carmine Vassallo et al. “A Tale of CI Build Failures: An Open Source and a Financial Organization Perspective”. In: *2017 33th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2017, pp. 183–193.
- [11] Carmine Vassallo et al. “Un-Break My Build: Assisting Developers with Build Repair Hints”. In: *Proceedings of the 26th Conference on Program Comprehension*. ACM. 2018, pp. 41–51.
- [12] Anunay Amar and Peter C Rigby. “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. 2019, pp. 140–151.

- [13] Louis G Michael IV et al. “Regexes Are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019.
- [14] Michael Hilton et al. “Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM. 2017, pp. 197–207.
- [15] Carmine Vassallo et al. “Every Build You Break: Developer-Oriented Assistance for Build Failure Resolution”. In: *Empirical Software Engineering* (2019), pp. 1–40.
- [16] Thomas Rausch et al. “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)*. IEEE. 2017, pp. 345–355.
- [17] Moritz Beller, Georgios Gousios, and Andy Zaidman. “TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration”. In: *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*. IEEE. 2017, pp. 447–450.
- [18] *Travis CI*. <https://travis-ci.org>. Accessed: 2019-10-25.
- [19] Taher Ahmed Ghaleb et al. “Studying the Impact of Noises in Build Breakage Data”. In: *IEEE Transactions on Software Engineering* (2019), pp. 1–14.
- [20] *Maven*. <https://maven.apache.org/>. Accessed: 2019-11-13.
- [21] *BART Jenkins Plugin*. <https://plugins.jenkins.io/bart>. Accessed: 2019-11-13.
- [22] *Travis Log Folds*. <https://blog.travis-ci.com/2013-05-22-improving-build-visibility-log-folds>. Accessed: 2019-11-18.
- [23] Serge Abiteboul. “Querying Semi-Structured Data”. In: *International Conference on Database Theory*. Springer. 1997, pp. 1–18.
- [24] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. “Characterizing Logging Practices in Open-Source Software”. In: *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 102–112.
- [25] Pinjia He et al. “Towards Automated Log Parsing for Large-Scale Log Data Analysis”. In: *IEEE Transactions on Dependable and Secure Computing* 15.6 (2017), pp. 931–944.
- [26] Meiyappan Nagappan and Mladen A Vouk. “Abstracting Log Lines to Log Event Types for Mining Software System Logs”. In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE. 2010, pp. 114–117.
- [27] Pinjia He et al. “An Evaluation Study on Log Parsing and Its Use in Log Mining”. In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2016, pp. 654–661.

-
- [28] Meiyappan Nagappan, Kesheng Wu, and Mladen A Vouk. “Efficiently Extracting Operational Profiles from Execution Logs Using Suffix Arrays”. In: *2009 20th International Symposium on Software Reliability Engineering*. IEEE. 2009, pp. 41–50.
- [29] Adam Oliner, Archana Ganapathi, and Wei Xu. “Advances and Challenges in Log Analysis”. In: *Communications of the ACM* 55.2 (2012), pp. 55–61.
- [30] Wei Xu et al. “Detecting Large-Scale System Problems by Mining Console Logs”. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM. 2009, pp. 117–132.
- [31] *Microsoft Program Synthesis using Examples SDK*. <https://microsoft.github.io/prose/>. Accessed: 2019-10-25.
- [32] Oleksandr Polozov and Sumit Gulwani. “Flashmeta: A Framework for Inductive Program Synthesis”. In: *ACM SIGPLAN Notices*. Vol. 50. 10. ACM. 2015, pp. 107–126.
- [33] Tom M Mitchell. “Generalization as Search”. In: *Artificial Intelligence* 18.2 (1982), pp. 203–226.
- [34] Vu Le and Sumit Gulwani. “Flashextract: A Framework for Data Extraction by Examples”. In: *ACM SIGPLAN Notices*. Vol. 49. 6. ACM. 2014, pp. 542–553.
- [35] *Excel FlashFill*. <https://support.office.com/en-us/article/using-flash-fill-in-excel-3f9bcf1e-db93-4890-94a0-1578341f73f7>. Accessed: 2019-11-18.
- [36] *Create Custom Fields in a Log Analytics Workspace in Azure Monitor*. <https://docs.microsoft.com/de-de/azure/azure-monitor/platform/custom-fields>. Accessed: 2019-11-11.
- [37] *PowerShell ConvertFrom-String*. <https://docs.microsoft.com/de-de/previous-versions/powershell/module/Microsoft.PowerShell.Utility/ConvertFrom-String?view=powershell-5.0>. Accessed: 2019-11-11.
- [38] Sumit Gulwani. “Automating String Processing in Spreadsheets Using Input-Output Examples”. In: *ACM SIGPLAN Notices*. Vol. 46. 1. ACM. 2011, pp. 317–330.
- [39] William R Harris and Sumit Gulwani. “Spreadsheet Table Transformations from Examples”. In: *ACM SIGPLAN Notices*. Vol. 46. 6. ACM. 2011, pp. 317–328.
- [40] Reudismam Rolim et al. “Learning Syntactic Program Transformations from Examples”. In: *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. IEEE. 2017, pp. 404–415.
- [41] Mohammad Raza and Sumit Gulwani. “Automated Data Extraction Using Predictive Program Synthesis”. In: *2017 31st AAAI Conference on Artificial Intelligence*. 2017.
- [42] Kevin Ellis and Sumit Gulwani. “Learning to Learn Programs from Examples: Going Beyond Program Structure.” In: *2017 26th International Joint Conference on Artificial Intelligence (IJCAI)*. 2017, pp. 1638–1645.
- [43] Tessa Lau. “Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI”. In: *AI Magazine* 30.4 (2009), pp. 65–65.

- [44] Robert C Miller and Brad A Myers. “Outlier Finding: Focusing User Attention on Possible Errors”. In: *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology*. ACM. 2001, pp. 81–90.
- [45] Mikaël Mayer et al. “User Interaction Models for Disambiguation in Programming by Example”. In: *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology*. ACM. 2015, pp. 291–301.
- [46] Alberto Bartoli et al. “Automatic Generation of Regular Expressions from Examples with Genetic Programming”. In: *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*. ACM. 2012, pp. 1477–1478.
- [47] Alberto Bartoli et al. “On the Automatic Construction of Regular Expressions from Examples (GP vs. Humans 1-0)”. In: *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*. ACM. 2016, pp. 155–156.
- [48] Stephen Soderland. “Learning Information Extraction Rules for Semi-Structured and Free Text”. In: *Machine Learning* 34.1-3 (1999), pp. 233–272.
- [49] Kathleen Fisher and David Walker. “The PADS Project: An Overview”. In: *Proceedings of the 14th International Conference on Database Theory*. ACM. 2011, pp. 11–17.
- [50] Qian Xi and David Walker. “A Context-free Markup Language for Semi-Structured Text”. In: *ACM SIGPLAN Notices* 45.6 (2010), pp. 221–232.
- [51] Kathleen Fisher et al. “From Dirt to Shovels: Fully Automatic Tool Generation from AD Hoc Data”. In: *ACM SIGPLAN Notices*. Vol. 43. 1. ACM. 2008, pp. 421–434.
- [52] Chia-Hui Chang, Chun-Nan Hsu, and Shao-Cheng Lui. “Automatic Information Extraction from Semi-Structured Web Pages by Pattern Discovery”. In: *Decision Support Systems* 35.1 (2003), pp. 129–147.
- [53] Dan Smith and Mauricio Lopez. “Information Extraction for Semi-Structured Documents”. In: *Proceedings of the 1997 Workshop on Management of Semi-Structured Data*. Citeseer. 1997.
- [54] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. “Introduction to Information Retrieval”. In: *Proceedings of the International Communication of Association for Computing Machinery Conference*. 2008, p. 260.
- [55] Dik L Lee, Huei Chuang, and Kent Seamons. “Document Ranking and the Vector-Space Model”. In: *IEEE Software* 14.2 (1997), pp. 67–75.
- [56] Annibale Panichella et al. “Parameterizing and Assembling IR-Based Solutions for SE Tasks Using Genetic Algorithms”. In: *2016 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE. 2016, pp. 314–325.
- [57] Giuliano Antoniol et al. “Recovering Traceability Links Between Code and Documentation”. In: *IEEE Transactions on Software Engineering* 28.10 (2002), pp. 970–983.

-
- [58] Andrian Marcus, Jonathan I Maletic, and Andrey Sergeyev. “Recovery of Traceability Links Between Software Documentation and Source Code”. In: *International Journal of Software Engineering and Knowledge Engineering* 15.05 (2005), pp. 811–836.
- [59] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. “Detection of Duplicate Defect Reports Using Natural Language Processing”. In: *Proceedings of the 29th International Conference on Software Engineering (ICSE)*. IEEE. 2007, pp. 499–510.
- [60] Gerard Salton, James Allan, and Chris Buckley. “Approaches to Passage Retrieval in Full Text Information Systems”. In: *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 1993, pp. 49–58.
- [61] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Pearson Education, 2010.
- [62] Shaun Phillips, Thomas Zimmermann, and Christian Bird. “Understanding and Improving Software Build Teams”. In: *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM. 2014, pp. 735–744.
- [63] Gerald Schermann et al. *An Empirical Study on Principles and Practices of Continuous Delivery and Deployment*. Tech. rep. PeerJ Preprints, 2016.
- [64] *PHPUnit Testing Output*. <https://phpunit.readthedocs.io/en/8.4/writing-tests-for-phpunit.html#testing-output>. Accessed: 2019-10-25.
- [65] *RSpec Output Format*. <https://relishapp.com/rspec/rspec-core/v/2-5/docs/command-line/format-option#documentation-format>. Accessed: 2019-10-25.
- [66] Frank McCarey, Mel O Cinneide, and Nicholas Kushmerick. “Recommending Library Methods: An Evaluation of the Vector Space Model (VSM) and Latent Semantic Indexing (LSI)”. In: *International Conference on Software Reuse*. Springer. 2006, pp. 217–230.
- [67] Tuomo Korenius, Jorma Laurikkala, and Martti Juhola. “On Principal Component Analysis, Cosine and Euclidean Measures in Information Retrieval”. In: *Information Sciences* 177.22 (2007), pp. 4893–4905.
- [68] *text2vec*. <http://text2vec.org/>. Accessed: 2019-11-10.
- [69] *Travis CI API*. <https://developer.travis-ci.org/>. Accessed: 2019-10-25.
- [70] Georgios Gousios. “The Ghtorrent Dataset and Tool Suite”. In: *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*. MSR ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 233–236.
- [71] *Travis CI Build Log Data Set*. <https://github.com/kth-tcs/travis-ci-build-log-dataset>. Accessed: 2019-11-21.
- [72] Benjamin Lorient, Fernanda Madeiral, and Martin Monperrus. “Styler: Learning Formatting Conventions to Repair Checkstyle Errors”. In: *arXiv preprint arXiv:1904.01754* (2019).
- [73] *GitHub*. <https://github.com/>. Accessed: 2019-11-21.

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