Bachelor Thesis

August 9, 2019

Build Log Differencing

More Efficient Failure Cause Identification

Noah Chavannes

of Vevey, Switzerland (16-701-872)

supervised by

Prof. Dr. Harald C. Gall

Dr. Sebastian Proksch

University of Zurich^UZH

s.e.a.l. software evolution & architecture lab

Bachelor Thesis

Build Log Differencing

More Efficient Failure Cause Identification

Noah Chavannes

University of Zurich UZH

s.e.a.l. software evolution & architecture lab

Bachelor Thesis

Author: Noah Chavannes, noah.chavannes@uzh.ch

Project period: 25. March 2019 - 25. September 2019

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Abstract

Continuous integration (CI) is transforming from a trend to the industry standard in software development [23]. A goal of CI is to develop software in a productive manner and with high quality. Building software is a sub-process of CI and does unfortunately not always succeed on the first try. Finding the failure cause in a broken build can be a tedious process and delay the work of other developers [25]. Comparing two successive build logs could yield the failure cause by highlighting the differences, but current text differencing tools fail to do so in an efficient manner. We introduce the concept of filtering build logs for irrelevant artifacts, which allows a meaningful comparison. Furthermore, we developed BLogDiff, a tool that applies the concept of noise filtering and allows developers to calculate the difference between two build logs. We evaluated the tool by conducting a survey and could show that the tool supports inexperienced developers in finding the cause of failing builds. Lastly, we analyzed the build log domain quantitatively and found different characteristics of successful and failing builds.

Zusammenfassung

Continuous Integration (CI) befindet sich im Wandel von einem Trend zum Industriestandard in der Softwareentwicklung [23]. Ziel von CI ist es, Software produktiv und mit hoher Qualität zu entwickeln. Das "Building" von Software ist ein Teilprozess von CI und gelingt bedauerlicherweise nicht immer auf Anhieb. Die Fehlersuche bei einem fehlerhaften Build kann sich als sehr mühsamer Prozess erweisen und die Arbeit anderer Entwickler verzögern [25]. Der Vergleich zweier aufeinanderfolgender Build-Protokolle könnte bei der Fehlersuche behilflich sein, indem die Unterschiede mit einem Textvergleichswerkzeug hervorgehoben werden. Aktuelle Programme unterstützen diesen Task jedoch nicht zufriedenstellend. Wir stellen das Konzept der Filterung von Build-Protokollen nach irrelevanten Artefakten vor, wodurch ein Vergleich ermöglicht wird. Darüber hinaus haben wir BLogDiff entwickelt, ein Tool, welches dieses Konzept anwendet und den Entwicklern ermöglicht, zwei Build-Protokolle zu vergleichen. Das Tool wurde durch eine Umfrage evaluiert. Wir konnten zeigen, dass das Tool unerfahrene Entwickler dabei unterstützt, die Ursache für fehlerhafte Builds zu finden. Abschließend analysierten wir die Build-Protokoll Domäne quantitativ und fanden verschiedene Merkmale, die erfolgreiche und fehlschlagende Builds unterscheiden.

Contents

Chapter 1

Introduction

A crucial part of developing software is converting source code into an artifact that can be delivered and executed on the target system of choice. This process is called "building" of a software project. There are many different building tools available for almost every programming language. One of the most popular tools to build Java projects is Maven.

Furthermore, there is an emerging trend in the software development industry, which is called Continuous Integration (CI) [23]. CI is the automation of processes that helps to build and verify software and is said to boost productivity and reduce the development costs on a project which explains its rise in the industry [23]. Building software is part of CI and should, therefore, not interfere with the productivity and cost benefits of CI practices. As long as there are no errors in the build process, that is not a problem. However, when the build process fails, it is crucial to find the cause of the problem quickly, as failing builds can delay the work of other developers and cause an unnecessary cost overhead [25,32,33].

Build tools like Maven produce an output during the process of converting source code into a deliverable. Such an output contains all sort of meta-information about the build process. These outputs are called build logs, and in the case of a failed build, reading them can help to find the cause of the failure. Finding failure causes can be a very easy and straight forward, but sometimes the failure cause is concealed in the details. Details in this instance could mean several thousand lines of information about a build and finding the cause of the failure, which could, e.g., be a version change in one dependency, can be a tedious process.

In this thesis, we propose a smarter way to find the error in the vast amount of log data, which consists of comparing the currently failing build log with the latest successful one. If a project is built in the scope of a CI setup, one should not expect too many changes between the build logs of two succeeding builds. When comparing these two logs with an ordinary differencing tool, the highlighted differences should yield the failure cause. There are already many differencing tools in existence (e.g., DiffViz developed by Frick et al. [22]), so why are they not used for comparing build logs and what is unique in the case of build logs that might be leveraged to improve the differencing? There needs to be a reason behind which we want to answer with our first research question.

RQ1: *How does build log differencing differ from ordinary text differencing?*

To answer **RQ1**, we analyzed why successive build logs cannot be simply compared. The reason we found were artifacts in build logs that change from one execution of a build process to the next execution but does not hold relevant information, which we defined as "noise". Examples of noise found in build logs are timestamps or download speeds. When comparing build logs that are unfiltered for noise, almost every line of a build log would be marked as changed when using an ordinary differencing tool and therefore render a comparison useless. To counteract this, we introduced the concept of filtering noise, which allows the comparison of two build logs in a meaningful way. We further classified different types of noise and developed a tool for filtering it.

An additional characteristic of build logs that favors specialized differencing tools over ordinary text differencing tools is the fact that build logs contain structured output. Specialized tools only need to compare equivalent blocks, whereas ordinary tools have to compare the whole file. This has led us to develop BLogDiff, a specialized differencing tool that allowed us to compare two build logs. In order to compare two logs, they first have to be parsed into a domain model. After successful parsing, we could calculate the difference between the two logs and save the result in the form of an edit script. The idea of an edit script is to store all edit operations that are needed to transform the first build log into the second one [37]. This allows deriving the content of the second build log from the first build log and the edit script. Additionally, analyzing the edit scripts yields information like the number of changes as well as their nature, which allowed us to have an in-depth look at the characteristics of build logs.

As we now could compare build logs, we could show that two successive and filtered build logs contained significantly fewer differences than comparing not filtered logs, which should help to highlight the actual differences.

Moreover, as we had the ability to compare filtered logs, we wanted to learn more about the build log domain-specific characteristics which yields the second research question.

RQ2: *What are the characteristics of a build log difference?*

To answer **RQ2**, we analyzed different datasets containing edit scripts of build log differences between two succeeding passing builds as well as from passing build followed by failing builds. We found that the similarity threshold, which decides whether two lines get marked as updated or as an addition in one log and a deletion in the other one, has an impact on the quality of the differencing. Furthermore, when not handling the output lines of downloading dependencies separately, the quality of the differencing decreases. We found that there are differences in the composition of edit scripts based on the outcome of the build and that there are only marginal changes when comparing two succeeding build logs. Lastly, there are differences in the distribution of failures within the build log when comparing passing builds and failing builds.

To validate if the concept of filtering noise is useful and whether comparing filtered logs helps a developer in his daily workflow, we wanted to answer the following research question.

RQ3: *Does a specialised differencing tool support a developer reading a build log?*

To answer **RQ3**, we developed a web application that integrates itself via a browser extension into the CI platform of our choice (Travis CI). With just one click, the developer can compare the current build log with its succeeding successful counterpart. We tracked how the user used the web application, and furthermore conducted a survey to measure the users' satisfaction.

The evaluation of the survey, the tracking data, and the comments from the community showed that the tool is especially useful for inexperienced developers. It helps them to find the failure cause more quickly and therefore provided them with an advantage and makes them more efficient. The experienced developers seem to agree that they do not benefit from using BLogDiff as they are quick in finding failures regardless of using a tool or not, which was consistent with the findings in our data.

The contributions of this thesis are the concept of filtering for noise, the conducting of the survey, a quantitative analysis of the build log domain as well as the development of a web application including a browser extension.

The datasets used, as well as the source code, are publicly available [20, 29–31].

Chapter 2

Build Log Differencing

After having established the importance of a quick failure cause identification, we present our solution approach to failing builds, which consists of comparing two successive build logs. We first discuss the limitations of ordinary text differencing tools and why they cannot be used to compare build logs. We introduce the idea of build log differencing and present a concept which overcomes the weakness of ordinary text differencing. Lastly, we show the core processes that are needed in order to convert our proposed approach into a usable tool, that supports developers in their daily workflow.

2.1 Concept of a Build Process

Continuous Integration Rausch, Hummer, Leitner, and Schulte show in an empirical analysis that failing builds especially pose a threat to the efficiency of developers within a Continuous Integration (CI) environment [32]. CI is a process that helps multiple developers that work on the same project to manage and merge their code. Usually, there is a rule that every developer has to commit and merge his code into the version-controlled repository of choice (e.g., Git). After the code is committed, automated tools verify and build the project and indicate whether a build was successful [28]. This ensures that there is a functioning code base that could be delivered at any time. Furthermore, CI is said to boost the productivity and quality of the software under development [23]. Among other things, this is why CI is becoming more and more the industry standard [23].

Travis CI There are several different tools that support the CI process. One of those tools is Travis CI [16]. Travis CI provides a charged platform for closed source projects as well as a free platform for open source projects. A study of Hilton et al. shows that out of 34,544 open source projects on GitHub, 90.1 % of the projects that use any CI, use Travis CI [23]. Other popular CI tools are Jenkins [6] or CircleCI [4]. Travis CI is widely customizable and supports a variety of programming languages. We decided to use Travis CI, as we needed as many freely available build logs as possible. Furthermore, it provides a public API which we could develop a client for, which allowed us to query information about builds.

Travis CI can be easily configured to monitor branches of GitHub [15] repositories. As soon as Travis CI detects a new commit, it executes all commands that are written in the mandatory *.travis.yml* file of a project. Most of the time, one of the commands is to execute a build. In Travis, there are builds and jobs, whereas a build can contain multiple jobs. A job is the execution of a script that, in our case, executes a Maven build.

Maven An important part of the CI process it the automation of the build process. Maven is primarily such a build automation tool [9]. Additionally, it provides metadata about a project and uniform guidelines for developers. It is one of the three most popular Java build tools [8].

Building is an essential process in software development and means the conversion of source code into an artifact, that can be delivered and executed on the target system. The process consists of multiple different tasks which can vary from build tool to build tool. In regards to Maven, the process includes validating the project, compiling the source code, and testing it. Furthermore, the compiled code is converted into a deliverable format (e.g., JAR or WAR) and some integration tests are run. The deliverable artifact is then installed into the local repository, which allows it to be used as a dependency for other local projects. In some cases, the deliverable is also deployed on the target system [10].

Maven uses a POM (Project Object Model), which is an XML file that contains all information needed to execute a build. The *pom.xml* file contains meta-information about the project itself as well as information about other projects/modules that are part of the project. Furthermore, it contains a list of all dependencies to third-party libraries used in the project, a list of plugins that can be executed in any phase of the build process as well as general build and environment settings [12].

Output of a Build Process During the build process, Maven produces an output containing information about the current stage of the process. This output is called a build log. These build logs are an object of investigation of this thesis. We want to understand the connections and peculiarities of build logs. What information can we extract from build logs that can help a developer troubleshoot? In order to analyze build logs, we need to understand the output. An explanatory build log is shown in Listing 2.1. For each module, the build log contains the output of all plugins that are executed as well as their goal. Furthermore, depending on the number of modules built, there is a reactor summary that summarizes the build process of each module. Finally, there is the final summary of the build stating total time and whether a build was successful or not.

Failing Builds An output of such a build process also indicates when a build process could not be executed successfully. Failing builds can block the work of many other people and cause delays, which directly translates into costs [25]. Finding the failure cause is not always a straight forward process. Applying a different strategy than manually analyzing the build log could help to save time in more complicated cases.

2.2 Text Differencing to Find the Failure Cause in a Failed Build

When a build fails, developers search for the failure cause by manually reading the build log. A possible alternative to manually analyzing build logs in more complicated special cases would be to compare successive build logs with each other. The study of Kerzazi, Khomh, and Adams mentioned that a common reason for failing builds are transitive dependencies [25]. Transitive dependencies are dependencies that are needed in projects a particular project depends on but are not directly used by this particular project [11]. An example of a more complicated build failure could be a change in a transitive dependency. The cause of the failure could not be directly deduced from the build log or the changes in the source code of the project. By comparing the currently failing build log with the previous successful build log, a developer could detect the change since the text differencing would show different version numbers when downloading

```
[...]
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Test Module 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] --- maven-enforcer-plugin:1.0:enforce (enforce-maven) [...] ---
[INFO] --- maven-source-plugin:2.1.2:jar-no-fork (attach-sources) [...] ---
[INFO] --- maven-install-plugin:2.4:install (default-install) [...] ---
[INFO] ------------------------------------------------------------------------
[INFO] Building Another Test Module 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] --- maven-enforcer-plugin:1.0:enforce (enforce-maven) [...] ---
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) [...]---
[INFO] --- maven-compiler-plugin:3.6.2:compile (default-compile) [...] ---
[INFO] Changes detected - recompiling the module!
[INFO] --- maven-install-plugin:2.4:install (default-install) [...] ---
[INFO] Installing another_test_module 0.0.1-SNAPSHOT.jar [...]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) [...] ---
[INFO] Tests are skipped.
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] Test Module ..................................... SUCCESS [ 14.020 s]
[INFO] Another Test Module ............................. SUCCESS [ 14.070 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 00:28 min
[INFO] Finished at: 2019-02-15T03:11:05Z
[INFO] Final Memory: 64M/735M
[INFO] ------------------------------------------------------------------------
```

Listing 2.1: Artificial example of a build log. The output of the plugins is not shown.

```
Line of 1st execution, no filter: "Downloading XY-0.0.1 (135 kB at 5.6 kB/s)"
Line of 2nd execution, no filter: "Downloading XY-0.0.1 (135 kB at 8.5 kB/s)"
```

Listing 2.2: Artificial example of the subsequent download of a dependency

these dependencies. There are already countless tools that allow comparing text line by line. Why aren't these tools used?

The reason for this is that build logs contain domain-specific artifacts that interfere with comparing a build log line by line. Listing 2.2 shows an example of such information. Even though the same dependency is downloaded (with the same size), the line would be marked at different as the average download speed did vary between both executions.

Noise in Build Logs These domain-specific artifacts that prevent the use of ordinary text differencing tools for comparing build logs, mostly contain meta-information about the build that differs from execution to execution but do not represent actual changes in the build process. Timestamps or download speeds are an example of such information that we define as *noise* going forward. This noise interferes with calculating the difference between two logs, as it is not a relevant difference but would still always be marked as such. We assume that filtering this noise would enhance the quality of the difference of two build logs as well as simplify the interpretation process of a differencing result, as only relevant differences are shown.

Build Log Differencing vs. Text Differencing After filtering build logs for noise, the calculated difference should be pure, free of disturbances, and only show the actual changes. This step of filtering noise is what allows the comparison of two logs, as comparing two unfiltered logs leads to a blurred difference since every line containing noise would be marked as a difference. This pure difference should now allow a search for failure causes that are not quickly found under normal circumstances. As mentioned before, each Maven build log can contain the output of multiple modules. As the outputs of the modules are not interconnected, one can look at each module individually. This enables us to only calculate the difference per module compared to ordinary text differencing where one usually has to search for changes in the whole file. The assumption that filtered build logs are comparable, and furthermore, that comparing build logs supports developers in finding failure causes has to be further investigated.

2.3 Proposed Approach: Build Log Differencing for Failure Cause Identification

Since we assume that noise filtering makes build logs comparable, and additionally, that comparing two build logs supports developers in resolving failures more quickly, we developed a tool that allows developers to perform build log differencing [2]. Figure 2.1 shows an overview of BLogDiff. The tool is based on a classical client-server architecture and is accessible via a browser extension. The general process flow and the important concepts are briefly described in the remainder of this section.

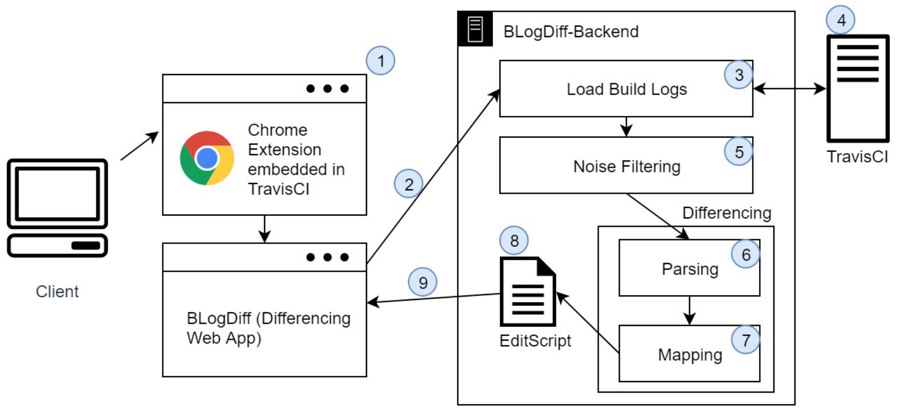


Figure 2.1: Overview of the build log differencing service

2.3.1 System Overview

The browser extension **(1)** embeds a button into the web interface of Travis CI. Clicking the button passes a *jobId* as a parameter and redirects the user to the BLogDiff web application. A request **(2)** from the web application triggers the server-side process, which commences with the loading of the build logs **(3)**. The BLogDiff backend requests the build logs from the Travis CI Server **(4)** via the Travis API. The build logs requested are the currently failing one and the precessing successful one. After loading the logs, the noise is filtered **(5)** in order to perform a meaningful differencing. Next, the build logs are parsed into a predefined data structure **(6)** that enables the service to compare both logs **(7)** and generate an *edit script*(**8)** out of the differences between the two logs. The BLogDiff web application then visualizes the result **(9)** by parsing the edit script and displaying the differences.

2.3.2 Preparation of Build Logs

In order to calculate the difference between the two build logs, they first need to be prepared accordingly.

Loading Logs The first step is loading the build logs form the Travis CI server. Travis provides a public API that can be queried by REST requests. In order to retrieve the build logs, we developed a client that was able to load information about builds, jobs, and repositories. Furthermore, an additional REST client to query GitHub was developed, which was necessary to validate if a repository actually contains a Maven project, since these are the only projects the tool currently can handle.

Noise Filtering Module The second step in preparing a build log for differencing is filtering the noise. After loading the build logs from the Travis servers, artifacts that interfere with the differencing are removed by chaining multiple filters that all implement the same filter interface. Each filter applies a set of rules to the raw build log and replaces the noise with some predefined tag. In the scope of this thesis, two filters were implemented. One that filters noise coming from Travis CI, and one that filters Maven related noise. Both filters can be easily extended, and additional filters could be added in the future.

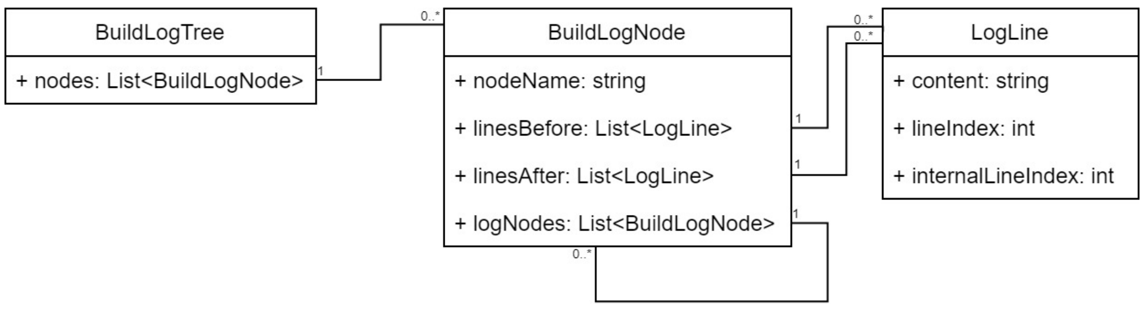


Figure 2.2: Meta model of the build logs

Parsing a Build Log After filtering a build log, it needs to be parsed into a format that can later be used for generating the difference between two build logs. Vassallo, Proksch, Zemp, and Gall presented a domain model that was specifically designed to hold build information about Maven builds [36]. As we designed our tool in a way, such that it is extendible for other programming languages, build tools, and continuous integration tools, we abstracted their approach and designed a more generic model which is shown in Figure 2.2. A build log representation contains a list of nodes. Each node has a unique name and a list of child nodes as well as a list of log lines before and log lines after the output of its child nodes. A *LogLine* object contains the content of its log line and the line index within the log (*internalLineIndex*), which is used by the differencing algorithm and serves as a position indicator for edit operations performed on the log line itself. Additionally, it contains the line number within the whole build log (*lineIndex*), which is only used to display the line number in the differencing web application. Listing 2.3 illustrates how an exemplary build log is mapped into the domain model shown in Figure 2.2. The outermost node is the *Travis* node. All log lines that are printed before the *Maven* node starts are mapped into the *linesBefore* list of the *Travis* node (*Lines 1-4*). *Line 4* marks the split between the *Maven* and the *Travis* node. The *Maven* node is added to the *logNodes* list of the *Travis* node. *Line 5-9* are the lines before the first module starts and are therefore mapped into the *linesBefore* list of the *Maven* node. *Line 10-16* are the output of a module, therefore a new node with the name *Module XY* is created and added to the *logNodes* list of the *Maven* node. Within that new node, all lines before the first plugin are mapped into the *linesBefore* list of the module (*Lines 10-15*). This process is repeated for every module and every plugin of the build log. The lines after the last plugin of the last module are added to the *linesAfter* list of the *Maven* node (*Lines 17-23*). Finally the lines after the *Maven* node are added to the *linesAfter* list of the *Travis* node (*Lines 24-26*).

Limitations The difficulty with parsing build logs is the fact that they do not come in a uniform format. For example, it is difficult to see where modules end and where a summary starts. Also, the boundary between Travis and Maven part is not easy to identify. There can always be new constellations that were not foreseen. Therefore we implemented a fallback parsing, which mapped all lines into a single node.

2.3.3 Generating the Edit Script

Once the two build logs are parsed into the build log domain model, the difference between them can be calculated. There are several approaches to calculating the difference between two logs. The BLogDiff backend is built in a way such that it can support several types of differencing algorithms. The difference is stored in the form of an edit script. The idea of an edit script is that

```
Build id: 555819511
Job id: 555819512
[...]
$ git checkout -qf 0dfdcdf823575256b3e6fab7d266d0b0aace2255
[...]
java version "1.8.0_151"
[INFO] Scanning for projects...
[INFO] Downloading from [...]
[...]
[INFO] -----------------------------------------------------------------
[INFO] Building Module XY 0.0.1-SNAPSHOT
[INFO] -----------------------------------------------------------------
[INFO] Downloading from [...]
[...]
[INFO] --- maven-enforcer-plugin:1.0:enforce (enforce-maven) [...] ---
[INFO] Plugin output line [...]
[INFO] -----------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] -----------------------------------------------------------------
[INFO] Total time: 00:28 min
[INFO] Finished at: 2019-02-15T03:11:05Z
[INFO] Final Memory: 64M/735M
[INFO] -----------------------------------------------------------------
[...]
Sending build context to Docker daemon 47.72MB
[...]
```

Listing 2.3: Artificial example of a Maven build log executed as a Travis job in order to illustrate the parsing.

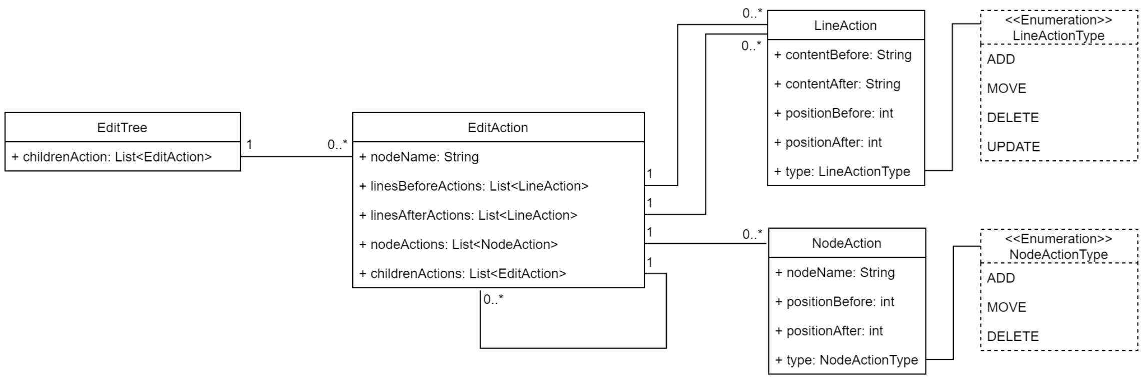


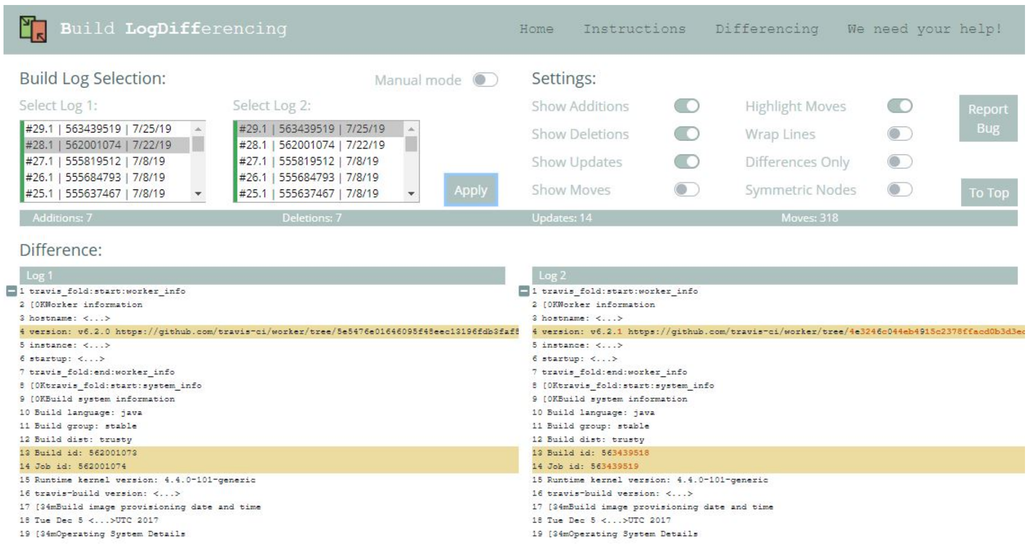
Figure 2.3: Domain model of the edit script

it contains all the changes that need to be applied to one build log in order to convert it into the second one. In other words, the second build log can be derived from the first build log and the edit script [37].

Edit Script In order to store information about the differences between two build logs, we introduced a further domain model of an edit script (Figure 2.3). The domain model of the edit script is inferred from the domain model of the build logs, that we have introduced previously (Figure 2.2). Each edit operation that is needed to convert one build log into another can be stored in the edit script domain model, which is achieved by a close coupling between the two domain models. The root of the edit script is the *EditTree* which is analogue to the *BuildLogTree* from the build log model. The *EditTree* contains a list of *EditAction* which represent the changes to a *BuildLogNode* of the build log model. Each node gets an own *EditAction* which is mapped using the unique node name as identifier. A *BuildLogNode* contains two lists with *LogLines*, the *linesBefore* and the *linesAfter* lists. When calculating the difference between two *BuildLogNodes*, only the *linesBefore* list of the first log has to be compared with the *linesBefore* list of the second log and the same for the according *linesAfter* list of both logs. This is due to the fact that lines not coming from the same location in a log do not have to be compared, as they are content-wise not correlated. For each difference between the log lines of two build logs, a *LineAction* object is created which contains information about the content before the change, the content after the change, the position before the change, the position after the change as well as the type of modification. Each modification can either be an addition of a line, a deletion of a line, an update of the content of a line or a move of the position of a line. The changes in the *logNodes* list of the *BuildLogNode* are mapped in the same way and stored as list of *EditActions* in the *childrenActions* list of the *EditAction*. Furthermore, there are also *NodeActions*, which store the information, whether the *BuildLogNode* as a whole was added, deleted, or moved.

2.3.4 Web application to Visualize the Differencing Result

After generating an edit script that contains all the operations that are needed to convert one build log into another, the build log difference has to be visualized. We built a web application that allows developers to calculate a difference between build logs and supports them in finding failure causes. Figure 2.4 shows the main interface of the BLogDiff differencing tool. The web application visualizes the differences by loading both build logs as well as the associated edit script. The build logs are displayed next to each other, and the edit script is used to highlight log



Select Log 1:
```
#29.1 | 563439519 | 7/25/19
#28.1 | 562001074 | 7/22/19
#27.1 | 555819512 | 7/8/19
#26.1 | 555684793 | 7/8/19
#25.1 | 555637467 | 7/8/19
```

Select Log 2:
```
#29.1 | 563439519 | 7/25/19
#28.1 | 562001074 | 7/22/19
#27.1 | 555819512 | 7/8/19
#26.1 | 555684793 | 7/8/19
#25.1 | 555637467 | 7/8/19
```

Show Additions, Show Deletions, Show Updates, Show Moves, Highlight Moves, Wrap Lines, Differences Only, Symmetric Nodes

Additions: 7 Deletions: 7 Updates: 14 Moves: 318

Difference:

Log 1
```
1 travis_fold:start:worker_info
2 [0KWorker information
3 hostname: <...>
4 version: v6.2.0 https://github.com/travis-ci/worker/tree/5e5476e01646095f48eec13196fdb3faf8
5 instance: <...>
6 startup: <...>
7 travis_fold:end:worker_info
8 [0Ktravis_fold:start:system_info
9 [0KBuild system information
10 Build language: java
11 Build group: stable
12 Build dist: trusty
13 Build id: 562001073
14 Job id: 562001074
15 Runtime kernel version: 4.4.0-101-generic
16 travis-build version: <...>
17 [34mBuild image provisioning date and time
18 Tue Dec 5 <...>UTC 2017
19 [34mOperating System Details
```

Log 2
```
1 travis_fold:start:worker_info
2 [0KWorker information
3 hostname: <...>
4 version: v6.2.1 https://github.com/travis-ci/worker/tree/4e3246c044eb4915c2378ffacd0b3d3ec
5 instance: <...>
6 startup: <...>
7 travis_fold:end:worker_info
8 [0Ktravis_fold:start:system_info
9 [0KBuild system information
10 Build language: java
11 Build group: stable
12 Build dist: trusty
13 Build id: 563439518
14 Job id: 563439519
15 Runtime kernel version: 4.4.0-101-generic
16 travis-build version: <...>
17 [34mBuild image provisioning date and time
18 Tue Dec 5 <...>UTC 2017
19 [34mOperating System Details
```

Figure 2.4: Main interface of the BLogDiff differencing tool

lines based on the edit operation performed on the line. An example of this process is shown in Figure 2.5. Two build logs contain the same node (*Node1*). The associated edit script also contains edit actions for *Node1*. The first build log contains two log lines, which are displayed in the output section of the first build log. The second build log only contains one line, which is displayed in the output section of build log two. Based on the edit actions contained in the edit script, all lines affected by an edit operation are highlighted. The *positionBefore* field always corresponds to the index of a log line in the first build log, whereas the *positionAfter* field represents the index of the log line in the second build log. The first edit action is an update action that affects the first log line of both logs. Therefore, both first lines are highlighted as an update (orange). The second edit action is a deletion, and the *positionBefore* field equals 2, which means the second line of the first build log was deleted (red). Therefore the line is marked as such in the first build log and no longer present in the second build log.

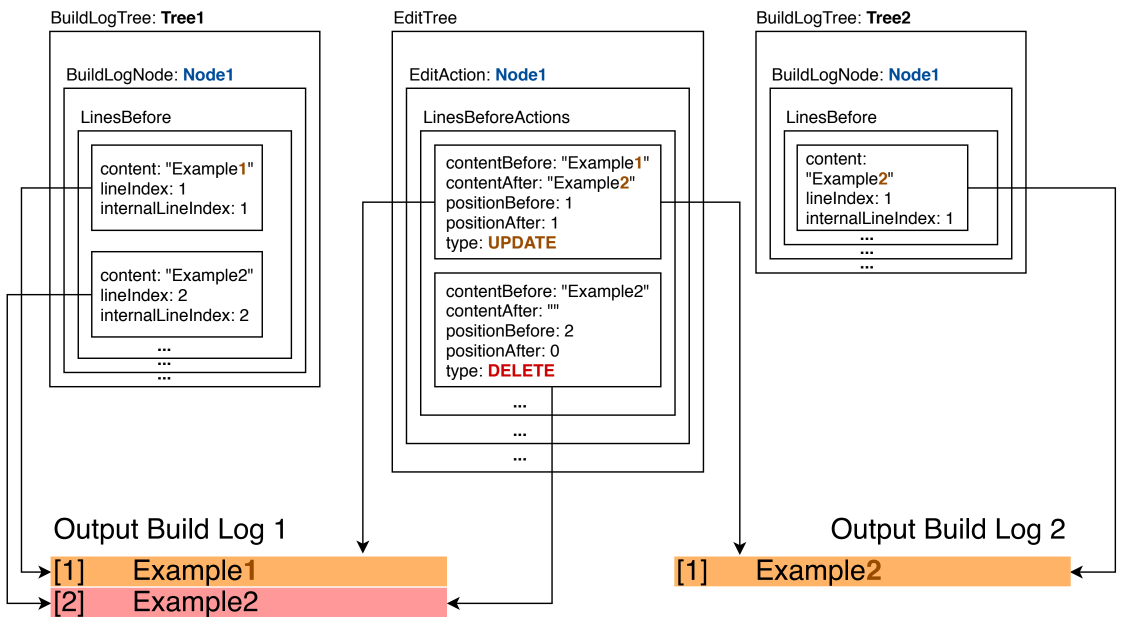


Figure 2.5: Exemplary visualization of an update and a delete operation, based on two build logs and the edit script

Chapter 3

Concretization of Concepts

In this chapter, we will reify concepts that were introduced previously. Since we assume that noise filtering makes build log differencing possible, we have to evaluate this claim and present our findings. Furthermore, we decide which differencing algorithm suits our tool and present the features of the web application.

3.1 Noise Filtering

In this section, we present our approach to solving the problem of noise in build logs.

Analysis Method We assume that many build logs share a common set of noise. Therefore, we developed a module that filters a build log for noise.

In order to classify noise, build logs needed to be compared manually. To do so, ten popular Maven projects that use Travis CI as a continuous integration solution were cloned and uploaded to our git repository and linked to Travis CI, so we could trigger builds ourselves. First, we selected a random project and built it with Travis CI. Then, without any changes, we built the project again. We assume that when we compare these two build logs, the differences should all classify as some sort of noise. We further assume that when filtered for noise, the two build logs should be almost identical. The only differences should be build-specific and relevant information like the ID of the build, for example. As at this point, no differencing algorithm was implemented, an online tool was used to visualize the differences [3]. This tool does not reliably map moved lines, which would distort the findings. In order to prevent this, the lines were sorted alphabetically and then compared. After doing so, the reference build logs that were not filtered had around 2,400 differences in 4,200 lines. Based on the differences found, we implemented a preprocessing module that filters the interfering artifacts found. We manually analyzed the artifacts and checked, whether they classify as noise. If they did, we extended our filtering module, such that it was able to filter the noise correctly. This process was repeated ten times with different projects while gradually extended the module at the end of each iteration. Eventually, the filtered build logs only contained predictable differences.

Findings The noise we found can be categorized into two different categories. The first one contains noise that is related to the continuous integration solution, Travis CI. The second category contains noise that is related to the output of Maven. Especially the Maven noise cannot be filtered conclusively, as there are endless plugins that generate a non-uniform output. A further problem is caching. If Maven is told to use the cache, all downloading related lines will be miss-

Name	Description	Example	Category
Travis CI environment	Information that is related to the environment the build process was executed on	hostname, instance, startup time, travis build version	Travis CI
Travis time	A start and end tag that is used to overlay durations of build steps in the Travis CI web build log visualisation	-	Travis CI
ANSI escape commands	Formatting for console commands	color coding	Travis CI
Git commands	Git related progress information	Counting objects, receiving objects, etc.	Travis CI
Download speed	Varying download speeds when downloading dependencies	-	Maven
Duration	Duration for sub processes	Time elapsed for tests, total time, etc.	Maven
Time	Timestamps of any sort	-	Maven
Progress	Progress of downloads and installations; often multiline	-	Maven
Maven process info	Maven outputs meta information regarding the build process	Final Memory usage	Maven

Table 3.1: Noise in build logs

ing which distorts the differencing. BLogDiff might be extended in the future to handle this issue. The noise found and implemented in BLogDiff is shown in Table 3.1.

Evaluation To verify if filtering for noise simplifies the process of reading build logs, ten random projects were selected and built twice without any changes. First, we applied our differencing algorithm to both logs without any filtering enabled and counted how many additions, deletions, moves, and update actions were present in the edit script. Then we applied the differencing algorithm to both logs with filtering enabled and counted the actions again. Filtering for noise should reduce the interference and enable a smoother differencing process. Therefore we expect the number of edit actions to decrease. The result of both approaches is shown in Figure 3.1. The findings are as expected. The total number of changes decreases when filtered for noise from an average of 59% to an average of 13% when compared with the total length of the build logs. We could even expect a bigger drop when implementing a more profound filter. What is noticeable is that the number of moves has increased. The reason for this are the lines that contain only small changes. Such lines are marked as an update without filtering, whereas they are identical after filtering, and are thus classified as equal or as move. Listing 3.1 shows an artificial example of such a change.

```
Line of first execution, no filter: "Downloading XY (135 kB at 5.6 kB/s)"
Line of second execution, no filter: "Downloading XY (135 kB at 8.5 kB/s)"
Line of first execution, filter: "Downloading XY (135 kB at <...>)"
```

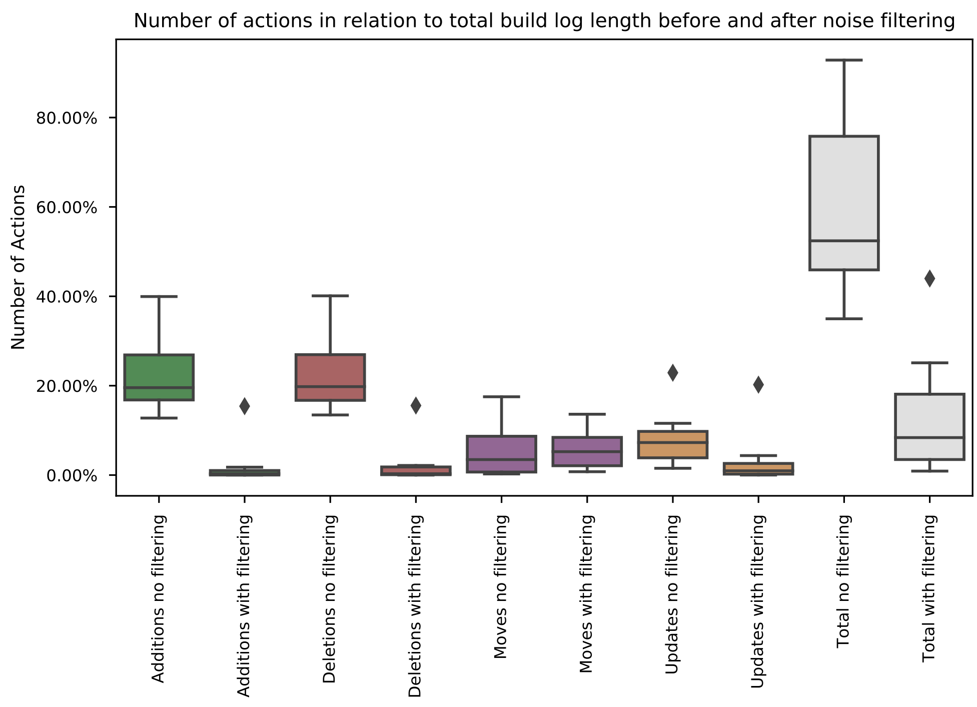


Figure 3.1: Impact of filtering for noise on the number of edit actions

```
Line of second execution, filter: "Downloading XY (135 kB at <...>)"
```

Listing 3.1: Artificial example of log lines that illustrates why there are more moves and less updates after filtering

Limitations In the analysis section, we stated that we used the build logs of 10 popular projects. We built a filter that handles the most occurring types of noise. This filter does not catch all noise but could easily be extended. The lack of ability to filter all sorts of noise is caused by the fact that the required implementation effort would exceed the scope of a prototype. The nature of noise, which can come in countless different forms and formats depending on the output of the endless plugins that are available, further complicates the filtering. This limitation does not pose a threat to validity, as a more profound filter would even emphasize our findings.

3.2 Differencing

The next concept for which we are evaluation possible concrete solutions is the differencing of two build logs. Differencing is the process of calculating the edit operations needed to convert a build log into another one. There are multiple approaches to calculating such a difference. In the scope of this thesis, we evaluated a line differencing and an AST differencing approach.

AST Differencing The first idea was to re-purpose the *GumTree* algorithm developed by Falleri et al. [21]. The purpose of their algorithm is to parse source code into an abstract syntax tree (AST) and calculate the difference between two versions of the same source code file. This algorithm was shown to be superior to a simple line-based approach when it comes to comparing source code, as it can more reliably detect updates and moves of code lines. Our idea was to re-purpose the algorithm and write an own parser, that maps our *BuildLogTree* into an object that can be fed into the *GumTree* algorithm.

Line Differencing The line differencing approach, on the other hand, uses an iterative approach. Each build log has a running line index where the two current lines of each build log are compared to each other. If the lines are equal, the index is increased, else the algorithm compares the similarity of both with the Levenshtein distance. If the similarity exceeds a certain threshold, the two lines are marked as updated. If a line is not present in the first log, it is marked as an addition. If a line is not present in the second log, it is marked as a deletion. All additions and deletions are stored in a map. In the end, the algorithm checks for matching addition and deletion pairs and marks them as a move.

Evaluation When comparing the performance of both approaches (Figure 3.2) one can see that the line differencing approach is superior to the AST differencing approach when it comes to the time needed to calculate the difference. In the reference build log used to measure the time, the line differencing approach took 1 second, whereas the AST differencing approach took 440 seconds. In contrast, the AST differencing approach generates a shorter edit script (-91 changes). However, this number should be interpreted with caution, as the AST approach often maps download lines wrongly as updates due to the high similarity of download lines which results in a lower amount of changes than mapping it as an addition and a delete as it is done by the line differencing approach.

We decided to map a build log into a tree-based domain model, but instead of using the corresponding AST approach, use the line differencing algorithm on the leaves, which represent the log lines of a build log. This allows us to maintain the advantages of a structured domain model while still benefiting from the speed of the line differencing approach.

Limitations Besides simplicity and speed, the line differencing approach also has its limitations, as it does not always map every line correctly. It is designed to work with a synchronous output of build logs where the order of the lines should not change from execution to execution. When it comes to the download lines of build logs, they sometimes switch position which lowers the quality of the differencing. The high similarity of all these lines further limits the ability to map every line correctly. We implemented countermeasures by looking for moved lines even though they should not exist in build logs. Furthermore, we implemented a separate, more restrictive similarity threshold for updates lines, which prevents wrongly marking download lines as updated. The limitations of the line differencing approach do not pose a threat to validity, as a more profound differencing algorithm would make the tool more effective.

3.3 The BLogDiff Web Application

The noise filtering concept, combined with the line differencing algorithm, was put in practice in the BLogDiff web application. The web application is developed using the Angular 7 framework [1] and sends all its requests to a Java Spring Boot backend [13]. This technology stack was chosen, as we already had experience with these frameworks, and they provide a straight forward approach to web application development. The BLogDiff web application was built as a prototype to evaluate whether such a tool can support a developer and to see if there is any demand for such a tool in the industry.

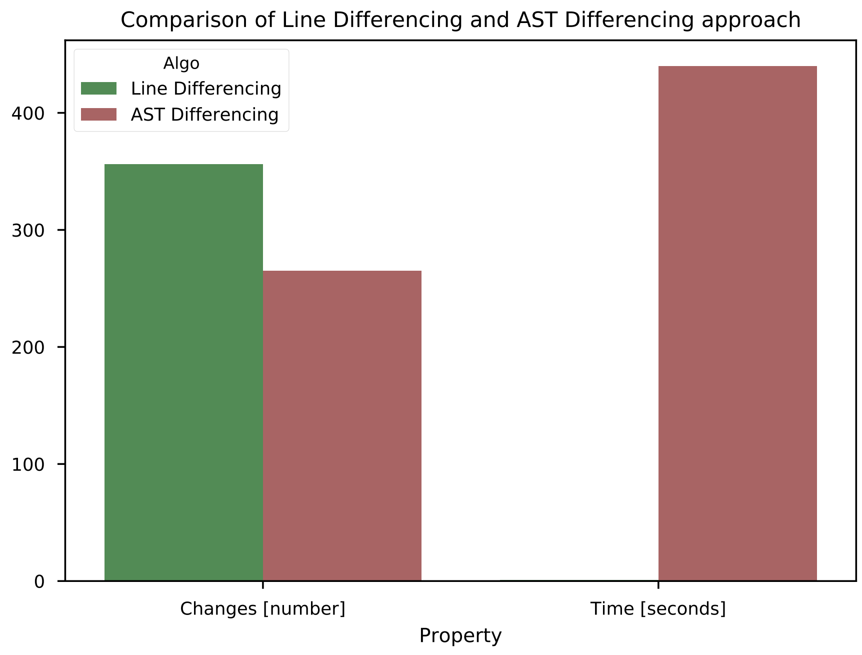


Figure 3.2: Performance comparison of the AST differencing approach and the line differencing approach

Intended Workflow The tool was developed with the idea that it should support but not hinder the developer in his everyday tasks. To achieve that, a Google Chrome extension embeds itself into the Travis CI interface. After building a project, a developer most likely checks the status of a build in the Travis CI interface. In order to access the tool if needed, the developer has to click on an icon and is redirected to BLogDiff. The icon does not distract but is clearly visible. Once the developer is redirected, the log of his current build, as well as the previous successful one, are loaded and directly displayed. The developer can now use BLogDiff to compare his logs in a more efficient way using all the features and the differencing provided by the tool. The differencing interface was shown in Figure 2.4.

3.3.1 Features of the Web Application

In this section, we describe and explain the features included in BLogDiff, which allow developers to perform build log differencing.

Job Selection The first step of performing a build log differencing is selecting the build logs that a developer wants to compare. Travis CI executes a Maven build process in the scope of a *job*. Therefore, in order to perform a differencing, two *jobs* containing build logs need to be selected. The available jobs are shown in a list, from which the developer selects the build logs. When using the Google Chrome extension, the job list, the current log, and the previous successful log are loaded automatically, and the differencing will be directly triggered (automatic mode of BLogDiff).

A developer can also use the tool in the manual mode. There are two possibilities to start a differencing in the manual mode. The user either enters a so-called repository slug (Figure 3.3),

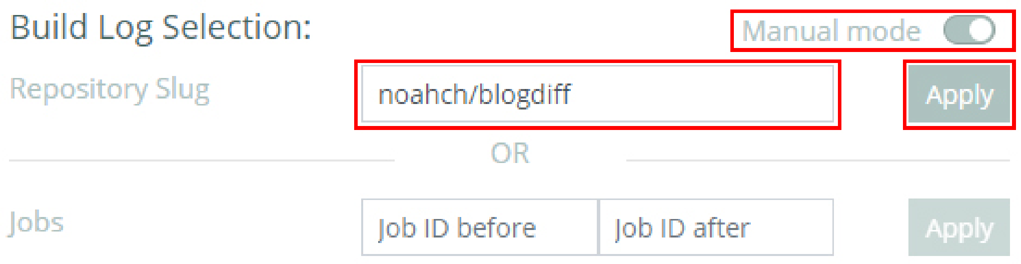


Figure 3.3: Loading jobs by entering a repository slug

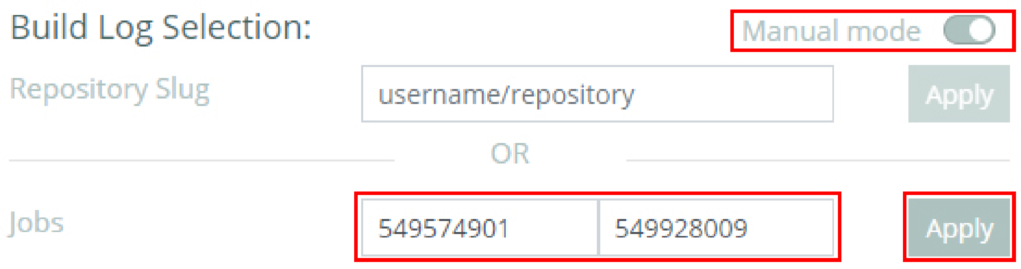
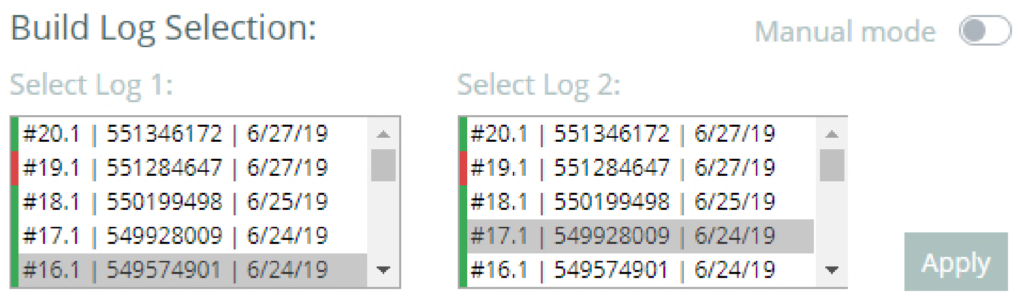


Figure 3.4: Manual selection of two build logs by entering the associated job IDs

which consists out of the username and the repository name of the project on GitHub or Travis CI, or he enters two job IDs manually (Figure 3.4). If the second option is chosen, the user has to ensure that both jobs are in the same repository.

After entering the repository slug or the job IDs, two lists containing the last 20 jobs of the chosen repository are shown in Figure 3.5. If the logs that should be compared are not present in the list, the developer has to enter the job IDs manually. The color on the left side indicates the status of a build (green=success, gray=cancelled, red=failed/errored). Further information, namely the build number, job ID, and the execution date are displayed. After selecting two jobs, the differencing can be calculated.



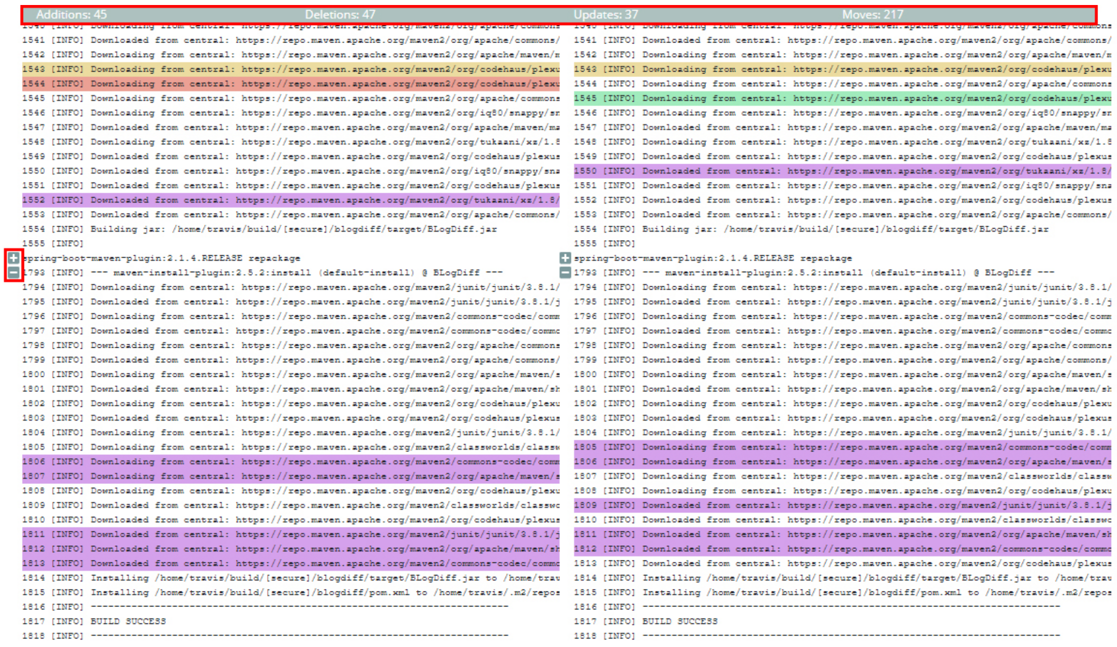
Select Log 1:

```
#20.1 | 551346172 | 6/27/19
#19.1 | 551284647 | 6/27/19
#18.1 | 550199498 | 6/25/19
#17.1 | 549928009 | 6/24/19
#16.1 | 549574901 | 6/24/19
```

Select Log 2:

```
#20.1 | 551346172 | 6/27/19
#19.1 | 551284647 | 6/27/19
#18.1 | 550199498 | 6/25/19
#17.1 | 549928009 | 6/24/19
#16.1 | 549574901 | 6/24/19
```

Figure 3.5: Selection of build logs based on a job



```
1541 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons/
1542 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/m
1543 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexu
1544 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexu
1545 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/apache/commons
1546 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/iq80/snappy/sn
1547 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/ma
1548 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/tukaani/xz/1.8
1549 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus
1550 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/iq80/snappy/sna
1551 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus
1552 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/tukaani/xz/1.8/
1553 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons/
1554 [INFO] Building jar: /home/travis/build/[secure]/blogdiff/target/BLogDiff.jar
1555 [INFO]
spring-boot-maven-plugin:2.1.4.RELEASE repackage
1793 [INFO] --- maven-install-plugin:2.5.2:install (default-install) @ BLogDiff ---
1794 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/junit/junit/3.8.1/
1795 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/junit/junit/3.8.1/j
1796 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/commons-codec/comm
1797 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/commons-codec/commc
1798 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/apache/commons
1799 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons/
1800 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/s
1801 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/sh
1802 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexu
1803 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus
1804 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/junit/junit/3.8.1/
1805 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/classworlds/classw
1806 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/commons-codec/comm
1807 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/s
1808 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexu
1809 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/classworlds/classwc
1810 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus
1811 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/junit/junit/3.8.1/j
1812 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/sh
1813 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/commons-codec/commc
1814 [INFO] Installing /home/travis/build/[secure]/blogdiff/target/BLogDiff.jar to /home/trav
1815 [INFO] Installing /home/travis/build/[secure]/blogdiff/pom.xml to /home/travis/.m2/repos
1816 [INFO] ------------------------------------------------------------------------
1817 [INFO] BUILD SUCCESS
1818 [INFO] ------------------------------------------------------------------------
```

```
1541 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons/
1542 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/m
1543 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexu
1544 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/apache/commons
1545 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexu
1546 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/iq80/snappy/sn
1547 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/ma
1548 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/tukaani/xz/1.8
1549 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus
1550 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/tukaani/xz/1.8/
1551 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/iq80/snappy/sna
1552 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus
1553 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons/
1554 [INFO] Building jar: /home/travis/build/[secure]/blogdiff/target/BLogDiff.jar
1555 [INFO]
spring-boot-maven-plugin:2.1.4.RELEASE repackage
1793 [INFO] --- maven-install-plugin:2.5.2:install (default-install) @ BLogDiff ---
1794 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/junit/junit/3.8.1/
1795 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/junit/junit/3.8.1/j
1796 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/commons-codec/comm
1797 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/commons-codec/commc
1798 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/apache/commons
1799 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons/
1800 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/s
1801 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/sh
1802 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexu
1803 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus
1804 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/junit/junit/3.8.1/
1805 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/commons-codec/comm
1806 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/s
1807 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/classworlds/classw
1808 [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexu
1809 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/junit/junit/3.8.1/j
1810 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/classworlds/classwc
1811 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/sh
1812 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/commons-codec/commc
1813 [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus
1814 [INFO] Installing /home/travis/build/[secure]/blogdiff/target/BLogDiff.jar to /home/trav
1815 [INFO] Installing /home/travis/build/[secure]/blogdiff/pom.xml to /home/travis/.m2/repos
1816 [INFO] ------------------------------------------------------------------------
1817 [INFO] BUILD SUCCESS
1818 [INFO] ------------------------------------------------------------------------
```

Figure 3.6: Differencing output

Differencing Once the differencing is calculated, the two build logs can be compared. An example of a build log differencing is shown in Figure 3.6. The top row shows basic statistics on the number of changes. The developer can also collapse individual modules to get a better overview. Additions of lines are highlighted in green, deletions of lines in red, updates of lines in yellow and moved lines in purple.

Settings The settings section of the tool allows a developer to manipulate the differencing in a way such that he can easily find the information he is looking for. The settings section is located in the top right of the differencing interface (see Figure 2.4). The settings are explained in the following list:

- **Show Additions:** Enable or disable the green highlighting of additions of lines.
- **Show Deletions:** Enable or disable the red highlighting of deletions of lines.
- **Show Updates:** Enable or disable the yellow highlighting of updates of lines.
- **Show Moves:** Enable or disable the purple highlighting of moves of lines.
- **Highlight Moves:** Enable or disable the highlighting of moved lines on click. When enabled and clicked on a move, the associated lines in the other log are highlighted with a lighter purple.
- **Wrap Lines:** Enable or disable the wrapping of lines.
- **Differences Only:** When enabled, only lines that have changed are shown.
- **Symmetric Nodes:** When enabled, only nodes (e.g., Maven modules) that are present in both logs are shown.

Gathering User Data BLogDiff tracks the activity of users using the web application. The data tracked included information about the repository and the build logs compared as well as the settings used and for how long a user is active. This data is saved in order to understand how the tool is used and to derive improvement measures.

Additional to the tracking, we also published a survey on the BLogDiff web application. We used an external library (surveyjs [14]) to implement the survey. This data gathered from the tracking and the survey should help us answering **RQ3**.

Other Features Apart from the differencing related functionalities, BLogDiff provides other useful features. A developer is able to file a bug report, which should help us to improve the tool further. Also, the functionality to enable or disable tracking is available. Users with privacy concerns are able to get their unique user ID and request the deletion of all their tracked data. Instructions and a demo of the tool are also available. Lastly, a developer can sign up to receive news about the progress of the tool.

Limitations As the scope of this thesis is to serve as proof of concept and to develop a prototype, the web application is currently limited to Java Projects build with Maven and using Travis CI as a continuous integration platform. The web application is built in a way such that it is easily extendible for other programming languages, build tools, and CI platforms.

3.3.2 Google Chrome Extension

As we already stated, the goal of BLogDiff is to be easily integrable in the developers' daily interaction with build logs. In order to access the web application when needed, a Google Chrome Extension was built for BLogDiff. The extension embeds itself directly into the web interface of Travis CI. This prevents that a developer has to navigate manually to a tool. Instead, the tool can be started directly with one click if needed but does not distort otherwise. The extension embeds a small icon next to each build job tab. By clicking on the icon, a new browser tab is opened, and BLogDiffdirectly generates a difference. To do so, it used the last previous successful job as a baseline for the comparison.
Additionally, the extension generates a unique user ID that is sent with every request, which allows us to track recurring users. Furthermore, the extension asks a user if he would like to partition in the survey.

Limitations The extension is only available in the Google Chrome browser. This does not pose a threat to the validity since no influence of the browser choice on the tool is to be expected.

Chapter 4

Characteristics of a Build Log Difference

We presented our build log differencing approach and could show that the concept of noise filtering indeed makes comparing build logs possible. Since we also selected a differencing algorithm and implemented a web application, we now have the tools to generate edit scripts. In this chapter, we look at a vast amount of edit scripts and quantitatively evaluate them in order to gain insights into the characteristics of build logs. We previously introduced **RQ2:** *What are the characteristics of a build log difference?*

We want to split **RQ2** into several sub-questions, which are all intended to contribute to the final answer of **RQ2**.

4.1 Similarity Threshold

When calculating the difference of two build logs, we want to know how similar two lines are to each other. In order to measure the similarity, we calculate a score based on the Levenshtein distance [7], which calculates the similarity of two strings. A score below a certain similarity threshold (e.g., score < 0.5) implies that less than 50% of one line has to be changed in order to convert it into the other line. If that is the case, we can mark a line as updated. If the similarity score of a line is above the threshold, it would be marked as an addition in one log and a deletion in the other log. This translates to two edit operations in the edit script instead of just one update action. Efficient differencing should aim for short edit scripts. Therefore it is important to not unnecessarily mark lines as addition and deletion, that could have been marked as updated. This leads to **RQ2.1**.

RQ2.1: *What is the impact of an similarity threshold for updates on the size and quality of the edit script?*

Methodology We define the size of the edit script as the number of edit operations contained in an edit script. In general, a shorter edit script is better than a longer one. So we could simply choose the threshold that results in the shortest edit script. In theory, a threshold of 1 would mean that two completely different lines would be marked as an update and lead to the shortest edit script possible. However, this would defeat the purpose of the differencing approach. The key is to find a balance between the length and quality of the edit script.

In several iterations, we will generate edit scripts of the differencing of the same set of build logs. Each iteration we change the similarity threshold. We will measure the size of the edit script by counting the number of edit operations contained in the edit script with different thresholds used. If a higher threshold yields a lower number of actions, we conclude that a higher threshold has a positive impact on the size of the edit script. If the size does not change or even increases, we conclude that a higher threshold negatively affects the size of the edit script.

Furthermore, we define the quality of an edit script as the amount of falsely positive mapped update lines. The less false positives are contained in the edit script, the better the quality.

We know from manually looking at edit scripts, that most false positives are due to download lines. Download lines are the lines printed during the download of dependencies, which occurs in almost every build log. Downloading is not necessarily executed in a synchronous fashion, which results in a changing order of the download lines from execution to execution. Furthermore, many different dependencies have very similar names, which can lead to false positives. Listing 4.1 shows such an example. *Line 2* in the first log, which was the download of the *spring-security-core* artefact, will be mapped as updated to *spring-security-web* in the second log (*Line 6*). This, on the other hand, results in a wrong mapping of *Line 3* as the *spring-security-web* was already marked as updated. To counteract this chain reaction, we want to evaluate, whether a separate threshold for the download lines, which is more restrictive than the threshold for all the other lines, helps to increase the quality of the edit script by reducing the number of false positives.

In several iterations, we will generate edit scripts of the differencing of the same set of build logs. Once with a separate download threshold, once without. Each iteration we change the similarity threshold. We will measure the quality of the edit script by comparing the number of updates and moves with and without a separate threshold for download lines. If for the same similarity threshold, the number of moves increases at the expense of updates when introducing a separate threshold for update lines, this concludes that there are fewer false positives. This would reflect the fact that update lines that were previously falsely marked as updates, are now correctly mapped as moved. Therefore, the quality of the edit script would increase.

Execution In order to conduct the experiments, we wrote a script that applies the BLogDiff algorithms to different build log pairs and generates edit scripts.

For the evaluation of the impact of a similarity threshold on the size of the edit script, we compared consecutive passing/passing build log pairs from 80 different open-source projects with each other. We compared them using five different similarity thresholds (0.1, 0.2, 0.3, 0.4, 0.5) which resulted in a dataset containing 1,253,490 data points, each representing a single edit operation.

For the evaluation of the effect of a separate download threshold, we used the same dataset as before, which did not account for a separate download threshold. Additionally, we applied the script on the same 80 projects with the same similarity threshold but adjusted our differencing algorithm to account for download lines by using a separate threshold. This resulted in a dataset containing 1,249,144 data points, each representing a single edit operation.

Results Figure 4.1 shows how the size of the edit script changes when applying different similarity thresholds. The data confirms the assumption that a higher threshold results in fewer changes seems to be true, even if only marginally. Therefore, we conclude that a higher similarity threshold has a positive effect on the size of the edit script. Increasing the threshold above 50% would further reduce the size of the edit script. We decided not to raise the threshold above 50%, as from a contextual point of view a line that changed more than 50% should not be classified as updated anymore, but rather as an addition and deletion of a line.

Furthermore, the data confirms that introducing a separate threshold for update lines increases the quality of the edit script, following the chain of arguments of the methodology section.

```
First execution
[INFO] Downloaded[...]spring-security-core-5.1.5.RELEASE.jar[...]
[INFO] Downloaded[...]spring-security-web-5.1.5.RELEASE.jar[...]
[INFO] Downloaded[...]spring-security-config-5.1.5.RELEASE.jar[...]
Second execution
[INFO] Downloaded[...]spring-security-web-5.1.5.RELEASE.jar[...]
[INFO] Downloaded[...]spring-security-config-5.1.5.RELEASE.jar[...]
[INFO] Downloaded[...]spring-security-core-5.1.5.RELEASE.jar[...]
```

Listing 4.1: Example of asynchronous download lines that are falsely marked as updates due to the high similarity of the names of related dependencies

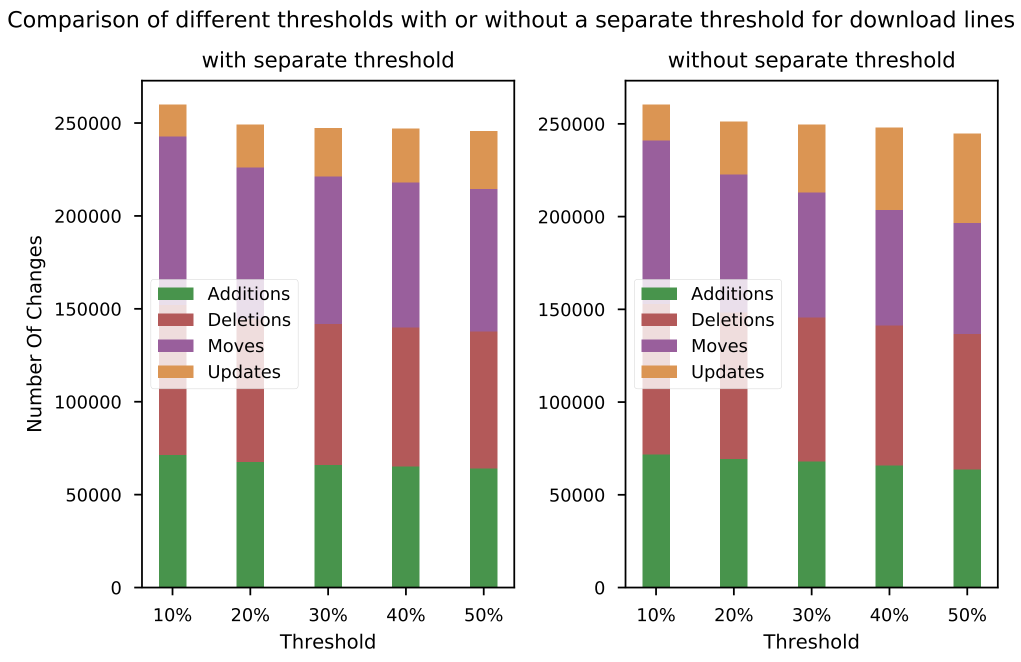


Figure 4.1: Composition of edit script with different thresholds and with or without a separate for download lines

Conclusion To answer **RQ2.1**, we conclude that the impact of different similarity thresholds on the size of the edit script is positive but marginal. Furthermore, a separate threshold for download lines leads to a higher quality of the edit script and less false positives.

4.2 Composition of the Edit Script

The execution of a Maven build with Travis CI can fail or succeed. Comparing the composition of the edit scripts of builds with different outcomes could help to uncover anomalies that cause build failures, which leads to **RQ2.2**.

RQ2.2: *How are edit scripts composed and how does the composition change based on the build status of the build logs under comparison?*

Methodology We define the composition of edit scripts as the relative amount of addition, deletion, update, and move operations contained in an edit script, compared to the total amount of operations. In order to compare the composition of edit scripts based on the build outcome, we investigate two types of build pairs. The first pair of interest is a passing build followed by a failing one, as this is the use case in which BLogDiff is intended to support the failure cause identification (*pass/fail* pair). The second pair of interest is a passing build, followed by another passing build (*pass/pass* pair). These *pass/pass* pairs represent a normal sequence of builds on a project where no build failure occurs. They are established as a baseline, and every deviation could be a possible failure cause. If the composition of *pass/fail* pairs differs from the composition of *pass/pass* pairs, we have to analyze the cause of the edit operations manually. A general assumption to be made is that *pass/fail* pairs should contain more delete operations, as a failing build automatically results in mapping all lines after the point of failure as deleted.

Execution In order to conduct the experiments, we had to generate edit scripts of both pairs. Our first dataset consists of several consecutive passing/passing build log pairs out of 80 different open-source projects. It contains 245,832 data points which all represent a single edit operation. The second dataset consists out of consecutive passing/failing build log pairs out of 49 different open-source projects and contains 222,837 data points which all represent a single edit operation. For both datasets, a similarity threshold of 0.5 was used.

Results Figure 4.2 shows the composition of edit scripts based on the two previously discussed sequences of interest. An edit script of a *pass/pass* pair mainly consists out of updates and moves. Only a few additions and deletions are present. In contrast, the edit script of a *pass/fail* pair mainly consists out of deletions, which is due to the fact that after a build fails in an early stage, all remaining lines of the passing build are marked as deleted. Furthermore, in a *pass/fail* pair are proportionally less updates present.

Table 4.1 shows the result of manually analyzing the nature of the changes per edit action type. Additions and deletions often come in pairs to represent updates that did not exceed the similarity threshold. Many of the deletions are due to failing builds where the remaining lines of the successful build cannot be mapped anymore. Moves are almost exclusively due to the asynchronous nature of downloading dependencies. Most updates are due to changing hashed, IDs, version numbers, file sizes or paths.

Conclusion To answer **RQ2.2**, we conclude that edit scripts are composed out of additions, deletions, moves, and updates and that the composition changes when comparing *pass/pass* or *pass/fail* pairs mainly in the number of deletions and updates.

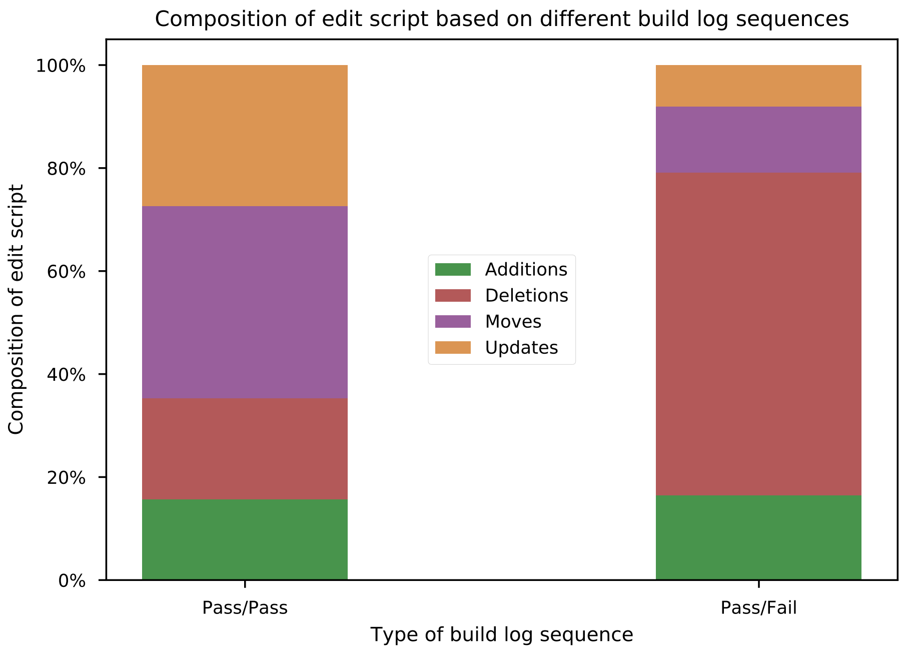


Figure 4.2: Composition of edit script based on different build log sequences

Edit Action	Nature of Change
Addition	New modules
	New plugins
	Update that did not exceed similarity threshold
Deletion	Deleted modules
	Deleted plugins
	Remaining lines after a failed build
	Update that did not exceed similarity threshold
Move	Order of downloading dependencies
Update	Hashes
	IDs
	Version numbers
	File sizes
	Paths

Table 4.1: Most common nature of changes per edit action we found in a manual analysis

4.3 Changes over Time

After being aware of how different build results affect the composition of edit scripts, we want to investigate how the compositions of edit scripts evolve over time. If we can show that the composition of edit scripts remains the same, we could make a point for the proper use of CI practices in the projects analyzed. This leads to **RQ2.3**.

RQ2.3: *How does the composition of the edit script evolve over time?*

Methodology In order to investigate the evolution of edit scripts, we need a series of successive build logs of the same project. We can then define the newest of these build logs as a baseline build log and calculate a difference between the baseline build log and each subsequent build log. The distance between two builds is defined as the number of builds that were executed between the two build logs under investigation, including the current one (e.g., the first and the third build of a series have a distance of 2). We also have to account for the different lengths of the build logs. We normalized the number of changes by dividing the number of changes by the length of the build log and then multiplying it by 100. Therefore each build log comparison can contribute at most 100 normalized changes. If regardless of the distance to the baseline build log, the composition of the edit script and the number of changes remains constant, we can assume that only marginal changes were made between both builds.

Additionally, instead of comparing each build log with the baseline build log, we compared them in pairs of successive build logs with a distance of 1. Analog to the first experiment, a constant composition of the edit script would suggest that only marginal changes were made between both builds.

We do not include moves in both experiments. As we established before, moves do not represent actual changes and would distort the findings due to the asynchronous nature of downloading dependencies that differs from execution to execution.

Execution In order to conduct the experiments, we chose 44 projects and selected a series of six successive build logs each. We then compared each build log of a series with the established baseline build log, which resulted in 823,084 data points which all represent a single edit operation. For the second experiment, we compared two neighboring build logs in pairs, which resulted in 727,423 data points, which all represent a single edit operation. We ensured that the time between all builds in a series is short enough, such that the series represents a common CI scenario, where the builds are triggered relatively close to each other. This also ensures that there are no major version changes, which would distort the findings.

Result Figure 4.3 shows the composition of the edit scripts as well as the normalized number of changes of the differencing between build logs of a series, once compared with a baseline build log, once compared in subsequent pairs. The composition of the edit scripts evolves with only small deviations in both directions, which means the changes between the build logs under comparison are marginal. This might be an indicator of the proper use of CI practices of the projects under investigation.

Conclusion To answer **RQ2.3**, we conclude that the changes over time stay relatively constant, as long as there are no major version updates.

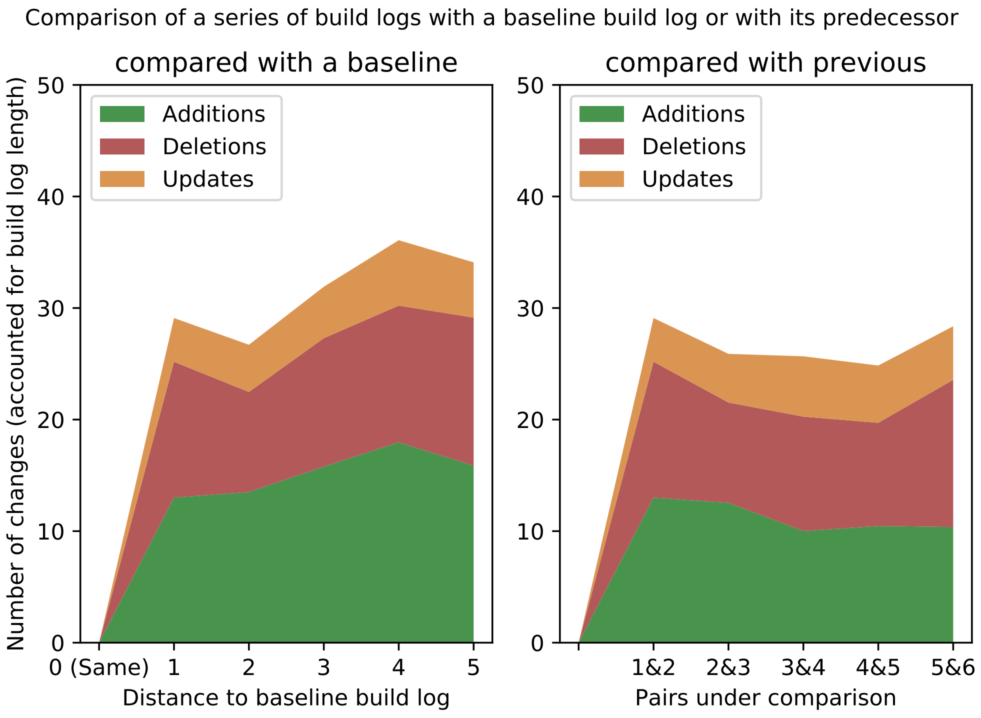


Figure 4.3: Composition of an edit script between a baseline build log and preceding build logs

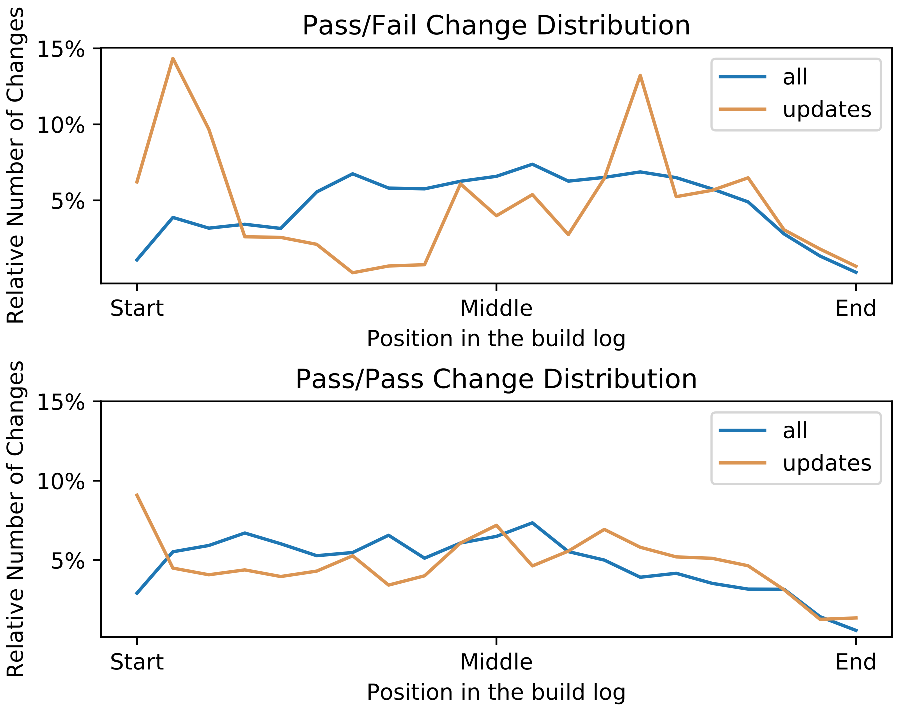


Figure 4.4: Distribution of all changes and of updates within the build logs

4.4 Distribution of Changes within a Build Log

To identify possible failure causes or typical behavior before a failure occurs, one could investigate where in the build log do changes occur and look for differences between *pass/fail* and *pass/pass* pairs. Possible deviations from *pass/pass* pairs could be indicators of a failure cause. This leads to **RQ2.4**.

RQ2.4: *Does the change distribution of pass/fail pairs differ from the change distribution of pass/pass pairs?*

Methodology In order to compare the change distributions between the two different build log pairs, we have to account for the different lengths of the build logs. We normalize the position of a change in the build logs by dividing the line index of the change by the total length of a build log. For each change, we can calculate the relative position in the build log. When plotting the aggregated number of changes of all build logs on the y-axis, and plotting the relative positions of the changes of a build log on the x-axis, we can visualize the distribution of the changes within a build log. When comparing the distributions between *pass/pass* pairs and *pass/fail* pairs, we can uncover anomalies. The cause for such anomalies has to be analyzed manually. As in previous experiments, we did not include moves, following the same arguments of irrelevance and inconsistency of download lines. We used the same datasets as in the experiment about the composition of edit scripts (Section 4.2).

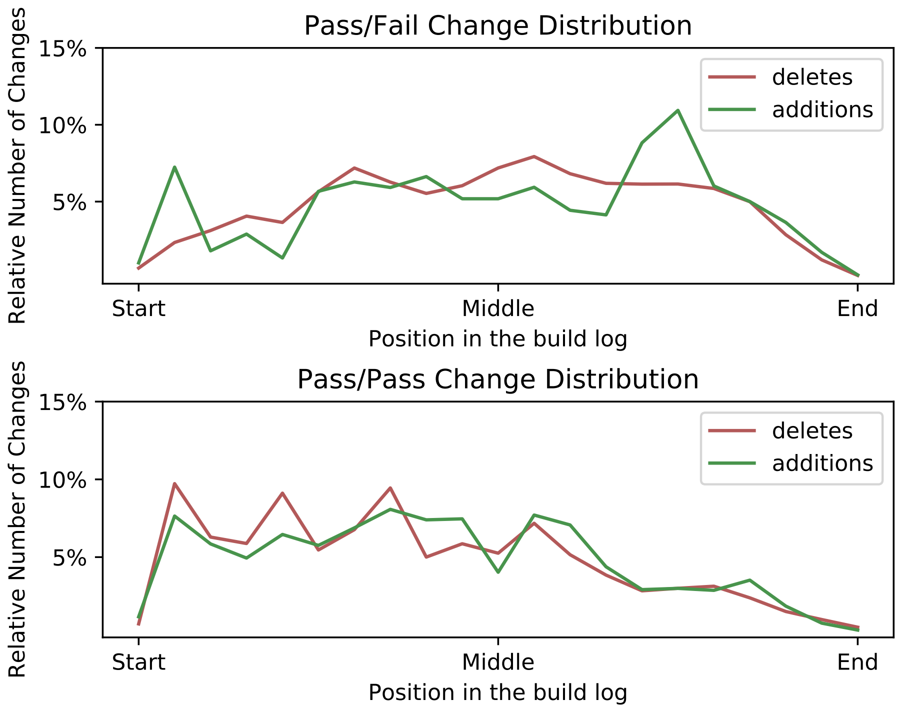


Figure 4.5: Distribution of additions, deletions, and moves within the build logs

Results Figure 4.4 shows that in general there are more changes at the beginning of a build log in *pass/pass* pairs than in *pass/fail* pairs in which the changes are mostly in the middle of the build log. An interesting observation was the fact that in *pass/fail* pairs, there are more updates at the beginning and at the end of a build log. One could argue for a correlation between a lot of updates at the beginning or near the end of the build log and a build resulting in a failure. The relatively large number of updates of a *pass/fail* pair are mostly due to changes in the execution of tests, the cloning of git repositories or changing hashes and IDs.

Figure 4.5 shows that *pass/pass* pairs can have many deletions at the beginning of the build log due to enabling of caching which makes the download of dependencies obsolete and therefore the associated download lines are missing in one log. In *pass/fail* pairs the deletions are more or less consistent, even though one would expect many deletions at the end of the build log since there is no more output on a build after it failed that could be compared with the previous successful one. However, because build logs fail at different points, the number of deletions is more or less evenly distributed across the scale. The peak in additions in *pass/fail* pairs is caused by the addition of the lines indicating that a build has failed. We did not include moves in both figures, as they primarily represent a change in the order of downloading dependencies, which is not of interest. A changing dependency, on the other hand, would be marked as updated and would very well be of interest.

Conclusion To answer **RQ2.4**, we conclude that the change distribution differs between *pass/-pass* and *pass/fail* pairs. The most important difference is the higher number of updates that *pass/-fail* pairs have near the beginning and the end of a build log.

Chapter 5

Usefulness of a Specialised Differencing Tool

As we have seen so far, build logs cannot be compared using ordinary text differencing tools. We have found different characteristics of *pass/pass* and *pass/fail* pairs of build logs, which indicates that comparing two build logs might help to find the failure cause.

Furthermore, we have previously introduced **RQ3:** *Does a specialized differencing tool support a developer reading a build log?*

To validate whether comparing build logs supports developers in finding the failure cause, we published a survey on our web application and invited developers to use our tool. In this chapter, we divide **RQ3** into several sub-questions and evaluate them.

5.1 Survey Design

In order to answer to **RQ3**, we designed a survey that consisted of 29 questions wherefrom four were open and 25 closed questions. We measured that it takes around three to four minutes to answer the survey, which we thought was short enough not to deter the users. The survey consists of four parts.

The first part is an introduction to the survey and is intended to give users the context they need to participate and understand the web application. It also allows us to ask the user whether he is already familiar with the tool and if not, show the instructions.

In the second part, we want to learn more about the users themselves. We are interested in the age and the gender as well as in the experience they have as professional software developers or in operations. Furthermore, we ask about the position within the hierarchy of their organization and their highest level of education. Lastly, we need to know how often the users are confronted with build logs and what their preferred programming language, build tool, and CI platform is.

The third part focuses on the current project of the surveyees. We want to know how many months a user is working on the current project and also how many other people are working on the same project. We ask about the role within the project (e.g., Senior Developer) and how many of their builds are failing. Lastly, we are interested in how long it takes them on average to find the cause of a failing build.

In the fourth part, we want to evaluate the tool itself. We ask several different questions about the usefulness, efficiency, and user-friendliness of the tool. In that context, we introduce three sub-questions that assist us to answer **RQ3**.

- **RQ3.1:** *Is BLogDiff perceived as useful by its users?*

- **RQ3.2:** *Is finding failure causes more efficient when using BLogDiff?*
- **RQ3.3:** *Is BLogDiff built in a way such as it is easy to use?*

For all closed questions, we either used a single select group, a number input field or a Likert-type rating scale [19]. This allowed us to make the results comparable without extensive data manipulation. Likert scales usually range from "Strongly Agree" to "Strongly Disagree" with the weaker forms of "Agree", "Neutral", and "Disagree" in-between [24]. For every question, we phrased the answers in a way such that they could easily be mapped to the standard Likert categories. As an example, the answers to the question *How would you rate BLogDiff in terms of usefulness?* were *very useful*, *useful*, *neutral*, *not useful* and *not useful at all*. This allows us to compare the answers to different questions. Table 5.1 shows a shortened list of the questions asked in the survey. All questions of the survey are available in an online archive. The complete set of questions, as well as the results, are available in an online archive [20].

Furthermore, we conducted a very small survey in the form of a pop-up which only asked whether the user would use the tool again.

5.2 Survey Evaluation

After designing the survey, it was published within the BLogDiff web application itself. We spread the link to the survey in several reddit communities that we thought could have an interest in such a tool. Furthermore, we published the idea in the forums of Travis CI and Google. In total, we had 38 participants whose survey answers are discussed in the remainder of this section.

5.2.1 Demographics

The average age of all participants was 31.1 years, whereas the youngest was 21 years, and the oldest 55 years old. Out of the 38 surveyees, 11 were female, and 27 were male. Furthermore, we asked for the highest level of education. Nine participants said they received some sort of technical or vocational training. Six participants graduated from college, nine people hold a bachelor's, and 14 people a master's degree. Nine surveyees stated that they are currently in the position of an assistant manager, whereas the rest staffed positions without management functionalities. The answers of the surveyees covered a wide spectrum when it comes to their experience as a professional in software development or operations. The average was seven years of experience with a standard deviation of 5.8 years. Interesting was that only three participants said they sometimes work with build logs. The rest did either work often or very often with build logs, which also qualifies all the answers as applicable as everybody is familiar with the build log domain to a certain extent. The primary language of all, but one participant was Java. The build tool of their preference was in 36 cases Maven, in once case Gradle and in one case an invalid answer. As for the choice of their preferred CI platform, 18 stated Travis CI, 16 Jenkins and 4 chose Circle CI.

On average the surveyees were already working for 11.2 months on their current project with a standard deviation of 9.9 months and a covered range of 1 month to 42 months. Furthermore, the average project had 7.4 other people working on the same project with a standard deviation of 4 people. Seven of the participants hold the role of a software architect on their project, whereas 18 were junior and 13 senior software developers. On average, 13.4% (std: 7.3%) of the builds fail, and the participants need an average of 2.7 minutes (std: 1.6 minutes) to find the cause of a failure.

Category	Question	Answer-Type
Demographics	9 questions [not listed here]	-
Project	How many of your builds fail (in percent)?	Number
Project	How long does it usually take you to find the failure cause when looking at a build log (in minutes)?	Number
Project	3 questions [not listed here]	-
Tool	How many of your builds fail (in percent)?	Likert-Type
Tool	How long does it usually take you to find the failure cause when looking at a build log (in minutes)?	Likert-Type
Tool	How would you rate BLogDiff in terms of usefulness?	Likert-Type
Tool	How would you rate BLogDiff in terms of its ease of use?	Likert-Type
Tool	Build logs contain noise (e.g., download speeds, timestamps, etc.) which is filtered by BLogDiff. Do you miss this kind of information?	Likert-Type
Tool	How would you rate BLogDiff in terms of effort needed to integrate it in your daily workflow?	Likert-Type
Tool	Based on your impression of BLogDiff, do you think the time to find the failure cause will change compared to your current method?	Likert-Type
Tool	Based on your impression of BLogDiff, how would you rate the accuracy of the differencing presented?	Likert-Type
Tool	Do you think BLogDiff provides an advantage compared to looking at a plain build log?	Likert-Type
Tool	How likely are you to use the BLogDiff (again)?	Likert-Type
Tool	How likely are you to recommend BLogDiff to a friend?	Likert-Type
Tool	Do you have any suggestions on how to improve BLogDiff?	Text
Tool	How do you usually process build logs?	Text
Tool	Do you have any other comments?	Text
Tool	3 questions [not listed here]	-

Table 5.1: Shortened list of the questions asked in the survey

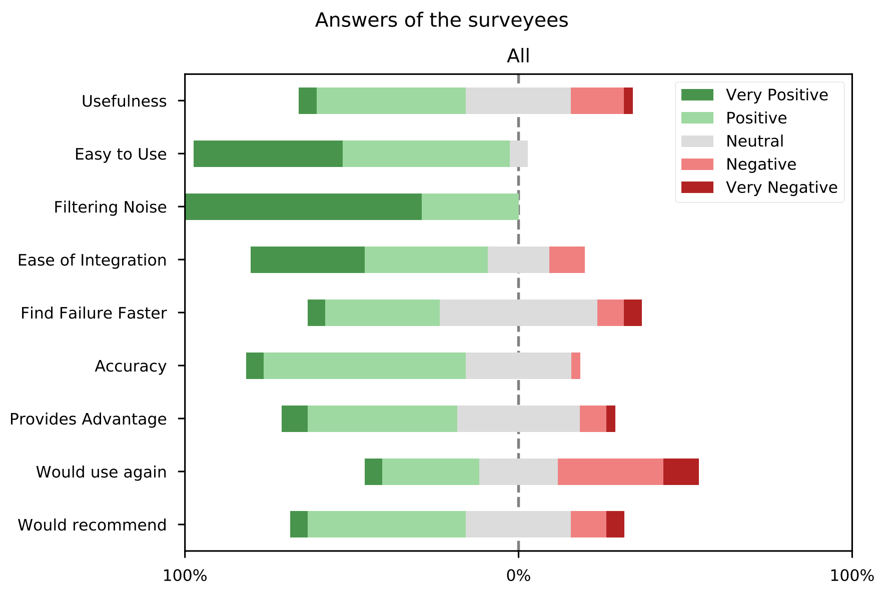


Figure 5.1: Overview of all the answers given by the surveyees

5.2.2 Tool Evaluation

The spectrum of all answers given by the surveyees is shown in Figure 5.1. It is important to note, that the answers on a Likert scale are ranked, but the interval between the categories cannot be considered equal [24]. Furthermore, out of the 38 participants, each participant answered all Likert-type questions we asked.

In general, the answers are more on the positive side of things. However, the users do not seem to agree when it comes to rating BLogDiff for its usefulness. Furthermore, more than half of the users would not use the tool again, which might already be the answer to **RQ3**. But we did investigate a little further and found some interesting facts that are discussed in the remainder of this section.

Usefulness of the tool First, we want to answer **RQ3.1:** *Is BLogDiff perceived as useful by its users?* To measure the usefulness of the tool, we will consider four metrics. The first question we asked was intended to figure out how the overall usefulness of the tool is perceived. Figure 5.2 shows that the surveyees predominantly gave positive feedback on this question, but there were also approximately 20% that did not find the tool useful in any way. A further interesting fact is that not one person participating in the survey said they would miss filtered information (noise). Since we are aware of the fact that our tool is just in the state of a prototype and that we will not possibly be able to predict all sorts of output from Maven, we asked whether the users perceived the differencing as accurate. Most surveyees agreed that the differencing result was accurate enough. Lastly, we asked if the users would recommend the tool or use it again. The answers to these two questions are more on the negative side, as only 34.2% said they would use it again, and 23.7% were indifferent. The question about recommending the tool paints a similar picture. These two answers differ from the general feedback the tool received. Furthermore, we looked at the data from the small survey that only asked whether a user would use the tool again.

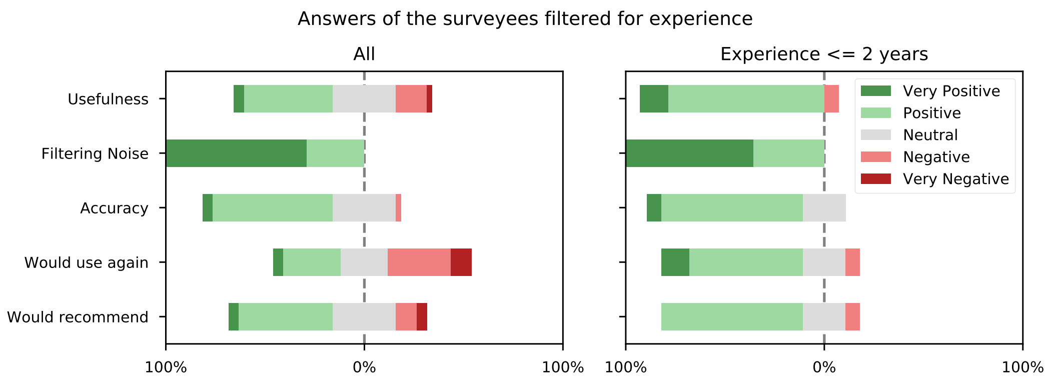


Figure 5.2: Overview of the answers related to the usefulness of BLogDiff and filtered for people with less than two years of experience

We had 20 participants in the small survey wherefrom 11 said they would use the tool again and nine said they would not, which is consistent with the data found in the big survey. We filtered the data of the big survey for people that do not more than two years of experience in professional software development or operations. The result is shown in Figure 5.2 and indicates that the tool better suits people that are not too familiar with the build log domain yet.

A further assumption could be that more experienced people face fewer problems with failing build logs and finding failure causes within. The regressions in Figure 5.3 endorse this assumption. We can show that older and more experienced people have less failing builds and do need less time to find the cause of a failure. Furthermore, we can show that the older and the more experience a user is, the smaller is the probability that he uses the tool again (Figure 5.4). This statement is further reinforced by the fact that more people think the tool is useful than there are people that would use the tool again (Figure 5.1). This means that some people think the tool is useful, even though they personally would not use it again, which leads back to the claim that the tool is better suited for inexperienced people. Further evidence that seems to emphasize this conclusion is the fact that almost all people rate the tool as easy to use. If this would not be the case, one could assume that the tool is built poorly, and therefore, people do not want to use it again.

We can now conclude **RQ3.1** by stating that the tool can support inexperienced users but is not suited for experienced ones, as they are already familiar in handling failing builds and finding failure causes quickly.

Efficiency of the tool Next, we want to answer **RQ3.2** *Is finding failure causes more efficient when using BLogDiff?* To measure efficiency, we asked the users if they think their time to find a failure will change when using the tool. As we already have established, more experienced users need less time on average to find a failure cause. Therefore we expect the more experienced users do not think their time to find a failure will reduce. Figure 5.5 confirms this expectation. In general, the findings are similar to the ones of **RQ3.1**. The less experienced users think they will be faster in finding failures when using the tool. Furthermore, we asked if the users think BLogDiff provides any advantage compared to looking at a plain build log. This does not necessarily mean finding the failure faster but can be anything from being able to collapse parts of the build log, having the differences highlighted, having metrics to the differencing or only displaying actual differences. As with the time to find a failure, the advantage of the tool was more valued by inexperienced

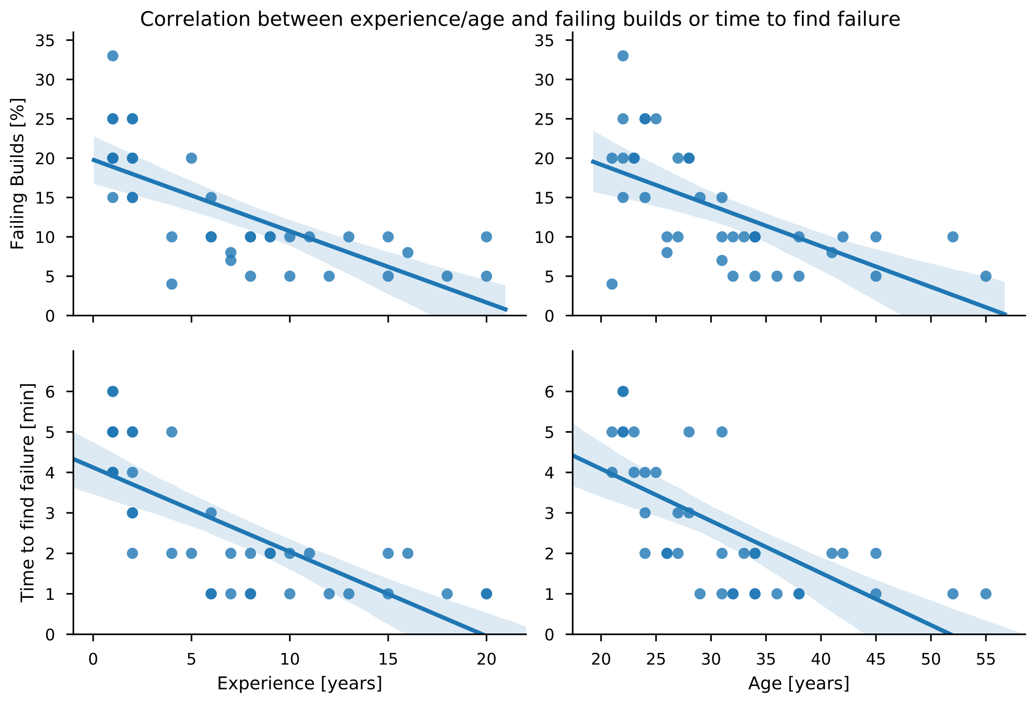


Figure 5.3: Correlation between experience/age and failing builds or time to find failure

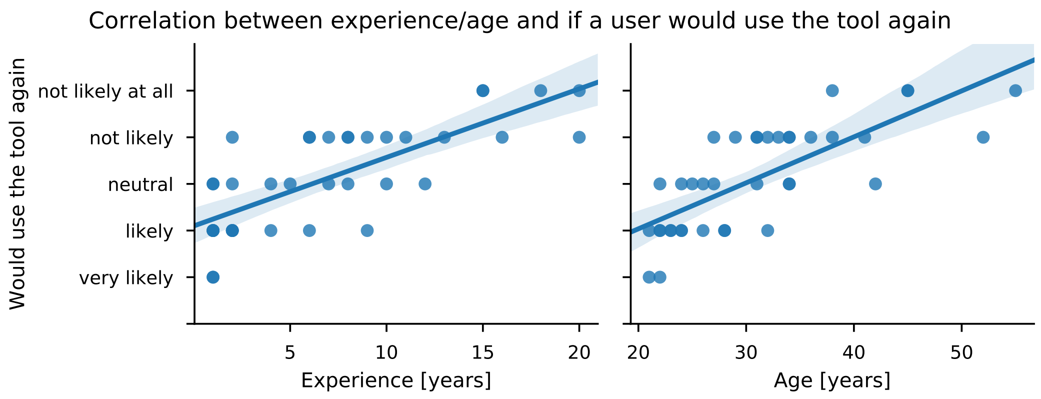


Figure 5.4: Correlation between experience/age and if a user would use the tool again

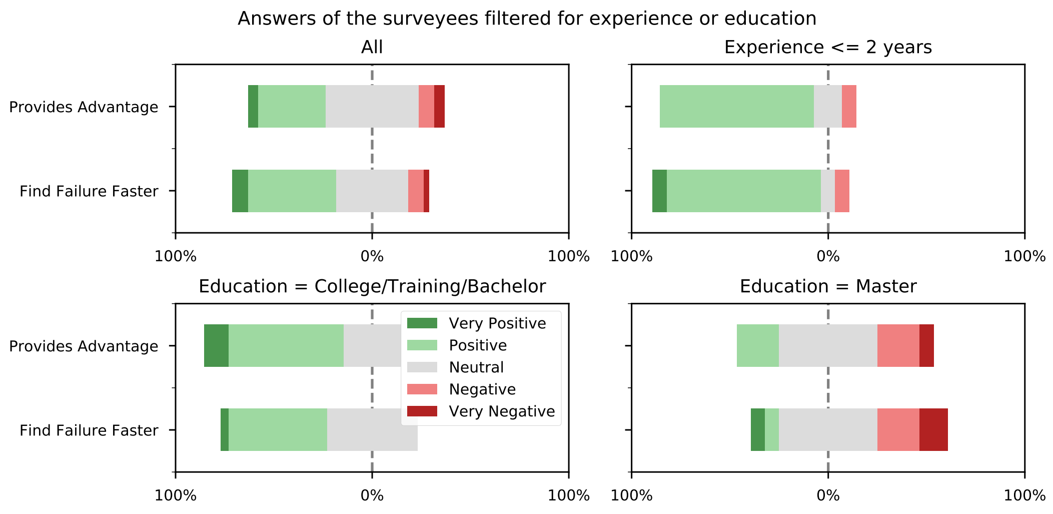


Figure 5.5: Answers of the surveyees filtered for experience or education

Level of education	Average time to find failure	Variance
Master	1.6 min	1.3 min
Technical/Vocational training	2.8 min	3.7 min
Bachelor	3.1 min	2.1 min
College	4.2 min	1.4 min

Table 5.2: Average time to find the cause of a failure grouped by highest level of education

users.

We also found that the highest level of education seems to correlate with the perception of the efficiency of the tool. People with a higher level of education are more experienced in general, which leads back to the regression discussed previously that shows the more experienced people are, the less time they need to find a failure. Indeed when comparing the average times to find a failure, the master degree holders lead against the ones holding a college, bachelor, or technical/vocational training degree (see Table 5.2).

We can now conclude **RQ3.2** by stating that finding failure causes with BLogDiff is efficient for less experienced people. Experienced users do not think that BLogDiff provides an advantage or reduced the time they need to find the cause of a failure.

User-friendliness of the tool Lastly, we want to answer **RQ3.3** Is BLogDiff built in a way such as it is easy to use? To measure the user-friendliness, we asked the surveyees how they would rate BLogDiff in regards to the ease of use and ease of integration into their daily workflow. The feedback to both questions was primarily positive, even though almost half the participants would not use the tool again (Figure 5.6). This concludes **RQ3.3** that BLogDiff seems to be built in a way such that it is easy to use.

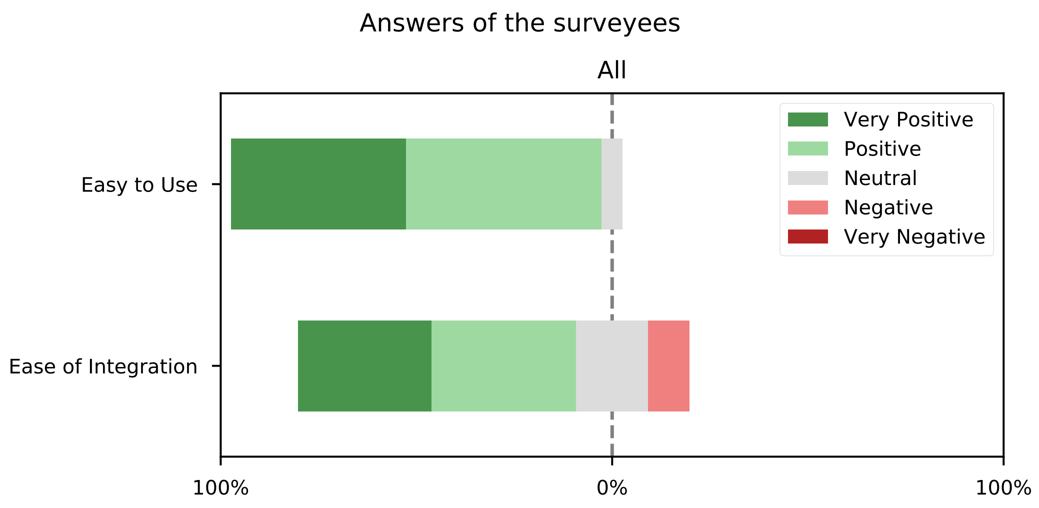


Figure 5.6: Answers of the surveyees in regards to user-friendliness

General Comments to the tool Additional to the closed questions, we also asked participants for a general comment on the tool. Four people stated that they think the tool is better suited for beginners, and someone mentioned that the tool only supports his process in edge cases. These comments are consistent with our findings. Furthermore, we wanted to know how they usually process build logs. Out of the 13 people that answered this question, all said they process the build logs manually, and three participants explicitly stated that they only look at build logs when a build fails. Lastly, we got two suggestions that campaign for a Jenkins [6] extension and one suggestion that would like to see support for other programming languages.

Additional Data From Tracking In order to obtain additional information to the questions asked in the survey, we also tracked the users' actions as long as they had the tracking settings enabled. We logged all activity in the form of a database entry but only as long as the user was actively using the site. We measured activity by observing for mouse clicks, scroll actions, and mouse movement. As long as one of the three activities could be observed, we wrote a log entry every 5 seconds. In the end, we could sum up the log entries per unique user and calculate the time he was actively using the web application. This allowed us to ensure that no idle time is present in the log entries. We found that out of the 63 unique users only three tested the tool on their own repository, the rest used the demo repository provided by us. On average, they used the tool for 3.1 minutes with a standard deviation of 5.3 minutes.

Chapter 6

Discussion of the Results

We have presented our approach to differencing and investigated the specific characteristics of the build log domain. Furthermore, we evaluated our tool for its right to exist. In this chapter, we want to compile our findings and reinforce them with voices from the community. Additionally, we discuss the threats to validity and propose future research topics related to this thesis.

6.1 Compilation of our Findings

So far, we have only answered sub-questions to the research questions we want to answer in this thesis. In this section, we will compile our findings and compose answers to each question.

The Difference between Build Log Differencing and Ordinary Text Differencing We introduced **RQ1** as *How does build log differencing differ from ordinary text differencing?*

To answer this question, we first had to find out why ordinary text differencing tools were not used to compare build logs. We found that one reason were artifacts that change from execution to execution but do not hold any relevant information, which we defined as noise. This noise is the reason why ordinary tools mark a vast number of lines as changed. We introduced the concept of filtering noise and could show, that when filtering build logs for noise, the number of changes reduces drastically. Therefore, we infer that filtering build logs for noise makes comparing of build logs possible. Furthermore, we found that the output of builds can be broken down into segments (e.g., the output of a module), so only the difference within and per segment has to be calculated, as the segments are not interconnected.

To conclusively answer **RQ1**, we state that build log differencing compared to ordinary text differencing has to account for noise. Additionally, build log differencing can use the structured output of build logs as an advantage, as the lines to be compared can be reduced by only comparing related segments.

Key Characteristics of a Build Log Difference After we had decided on our approach to build log differencing and implemented a web application, we could investigate the build log differencing domain by analyzing a vast amount of edit scripts. Our goal was to find answers to **RQ2** that we had introduced as *What are the characteristics of a build log difference?*

Our key findings that conclude **RQ2** were that the higher the similarity threshold is, the fewer edit operations are contained in the edit script. We found that a separate threshold for download lines is needed, as not having one would result in falsely marking download lines as updated, when in fact they just moved. Furthermore, we found that the composition of the edit script hugely differs between *pass/pass* and *pass/fail* pairs mainly in the number of update and deletion operations contained. The composition of an edit script stays relatively constant when compared over time, as long as there are no major version changes. Lastly, the distribution of changes differs between *pass/pass* and *pass/fail* pairs primarily in the number of updates near the beginning and the end of a build log, which could indicate anomalies that can cause a build to break.

BLogDiff is a Useful Tool for Beginners Once we had conducted the survey, we could analyze the answers from the community, which together are able to answer **RQ3**: *Does a specialized differencing tool support a developer reading a build log?*

In order to answer **RQ3**, we decided to evaluate the tool against the criteria of usefulness, efficiency, and user-friendliness. Additionally, we analyzed the user data we tracked and the comments to the open questions of the survey. Based on the answers of the surveyees, we found that especially the inexperienced developers rated the tool as useful and would use it again. The more experienced users seemed to agree that they do not need a specialized tool for analyzing build logs, most of them even agreed on the concept of filtering noise. We found a correlation between the age and experience of a developer and the time to find a failure cause. The more experienced a developer is, the less time he needs to find the failure cause. Furthermore, we found that also the highest level of education has an impact on the time needed to find a failure cause. We could show that inexperienced people stated, a specialized tool supports them in finding failure causes more quickly. Therefore we conclude that the tool helps to increase the efficiency of inexperienced developers. Lastly, the data of the survey shows that BLogDiff is built in a userfriendly way, which concluded our last criteria. Evaluating the open questions yields, that some users would endorse an extension for other languages and CI tools.

6.2 Reflection on BLogDiff

Since we have answered our research questions, we primarily want to reflect on our findings in the context of Continuous Integration. We also want to reflect on some reactions and voices of users of the communities, in which we promoted out tool [5].

Productivity-Enhancing Tool By now, we have established the importance of CI in the software industry. We are aware of the costs that are associated with failing builds and hindering other developers from continuing with their work [23, 25]. Additionally, there are studies that show that the quality of software improves when CI processes are in practice [35]. Therefore, embracing the CI process and being able to avoid delays, is a characteristic of BLogDiff that confirms its right to exist, especially when used by inexperienced developers. The most common causes for failing builds seem to be dependency related [25, 32, 34]. Having a tool that is able to detect such anomalies can be essential, as noted by *reddit_user_1*:

> *"I'd give some credit to the author. You may see the error message right away, but I think this tool could be able to trace first abnormalities in a process that led to the failure. One thing is seeing the issue; other is seeing the reason for it."* (reddit_user_1)

> *"It sounds a novel idea, but I'm afraid I can't see much real-world use, for me it's about failing fast and breaking the build into small components so when that component in the build fails it's easier to identify why things went wrong. If there's lots of unimportant information in the build logs, it would probably be better to reduce that logging level to error detail only rather than info or debug which adds a lot of the fluff to logs."* (reddit_user_2)

Reddit_user_2 proposed a different approach. Instead of filtering noise, *reddit_user_2* would reduce the log level to only show errors. This would indeed remove some noise, but this defeats the purpose as we want to detect anomalies in the details of a build log, where one would usually not search for failure causes. We claim that our approach to noise filtering is justified, especially because all of the 38 survey-participants agreed that they would not miss filtered artifacts in their build logs.

Educational Tool Besides the productivity-enhancing qualities of BLogDiff it can also serve as an educational tool. We have shown in our data, that inexperienced developers think, their time to find failures would decrease when using the tool. *reddit_user_3* seems to agree that the tool is suited for beginners than for experienced developers:

> *"This [finding failures in build logs] is not a problem as far as I can tell for the experienced folks. Maybe newer people have more difficulty finding the failure message. If you work on any complex system for a long time, finding the error messages is second nature and probably the easiest part of troubleshooting."* (reddit_user_3)

Reddit_user_3 makes an interesting point by stating that finding failure causes becomes second nature, once a developer is experienced enough. Here we think BLogDiff validates its right to exist as an educational tool, as the highlighting of differences might help inexperienced users to develop a sense for build logs and the nature of build failures.

> *"Honestly, I can't say that this would be very useful to me, and to be frank, I can't think of any instance where I've found myself going through past logs to identify what went wrong on the current one. [...]"* (reddit_user_4)

We also received comments from users, that did see any reason for them using the tool (*reddit_user_4*). As we received this answer in the DevOps community, in which the members are usually experienced, we rate this comment as further evidence, that the tool is less attractive to experienced developers. However, we cannot interpret this statement with absolute certainty.

Future Work In the scope of this thesis, we developed a prototype and made a proof of concept. To provide further benefits to the community, one could extend the tool in a way, such that it supports more programming languages and CI tools. The study of Beller, Gousios, and Zaidman shows that the languages with the most builds per project on Travis CI are Pearl and C++ [18]. These two languages could be a possible point of entry for future work. They further state that the most common cause of failures is related to test case execution. Therefore, the tool could also be extended with a feature that directly highlights the name of the test case that caused the failure.

> *"Does this only work with travis.ci? We have enormous build logs with many points of failure, but we'd want to use something like this offline."* (reddit_user_5)

Reddit_user_5 from the community asked for an offline version. This might not leverage the advantages of the combination with a CI tool anymore, as the build logs are not loaded automatically any longer, but could still help users in some unique cases.

In the Travis forum, [17] one user points out, that the problem is very situational and that a parser needs to be able to read the output of every tool in existence correctly in order to solve the problem. He further highlighted the importance of making the noise filter fully configurable and exportable. At the current state of our tools, the filters are extendible but cannot be imported, exported, or configured per repository, which could be another point of entry for future work.

Lastly, one could investigate why experienced developers do not seem to benefit from BLogDiff.

6.3 Threats to Validity

After having reflected on BLogDiff, we shortly want to address issues that might pose possible threats to the validity of this thesis.

Algorithms Due to the not uniform nature of build logs, it is impossible to predict every variation of noise that could be present in a build log. As we had limited resources, we only sampled ten projects that we could use for the noise classification. Having a larger sample size would ensure more precise filters and therefore increase the performance of the tool. The same is applicable for the parsing of the build logs. We could only build a ruleset for the parsing process that did hold in the examples we used. To counteract problems while parsing, we implemented a fallback mechanism. Again having a more sophisticated parsing algorithm would increase the performance of the tool.

Sample size When measuring the characteristics of build logs, we only had a limited sample size at our disposal. As we already established, the output of build tools is not always predictable. Therefore, a differencing between two logs that were not correctly parsed or filtered could distort the findings. The different lengths of build logs could be a further disturbing factor. To counteract this, we made sure every build log is weighted equally, so the chance of distortion is minimized.

Selection bias As our tool is a prototype and currently only supports Java as a programming language, Maven as build tool and Travis as CI platform, we only posted the link to our survey in communities related to it. The most submissions came from the DevOps community, which were mostly experienced people. As we found our tool is more suited for inexperienced people, we tried to balance the submissions from the experts by asking more students to participate in the survey. Moreover, we tried to keep all questions as neutral as possible in order to avoid a bias in the answers.

Chapter 7

Related Work

In this chapter, we will focus on publications that provided helpful insights and ideas in writing this thesis, and in general, focus on similar problems or propose different solution strategies.

Differencing Macho, McIntosh, and Pinzger developed a tool called BUILDDIFF, which is an approach to extract build changes from Maven build configuration files (pom.xml) [26]. Furthermore, they classified the changes into 95 different categories. By manually evaluating the changes in 400 build changing commits, they could show that their tool classified the changes with a precision of 96%. They found that the top ten most frequent changes cause 73% of the build changes. Some of these top ten changes include dependency changes and version number changes of the project or the parent of the project. Additionally, they could show that the changes are not evenly distributed in the project's timeline, but rather are more frequent around releases. BUILDDIFF relates to BLogDiff in the fact that both tools try to support developers in the process of resolving build fails by highlighting differences. However, the tools differ in their approach. BUILDDIFF focuses on showing the differences within the configuration files of a build, whereas BLogDiff focuses on the differences in the output of the build process itself.

Falleri et al. presented a new algorithm for source code differencing called GumTree [21]. Their algorithm parses source code in an abstract syntax tree and computes an edit script that also includes move actions instead of only additions and deletions as most other differencing algorithms do. They automatically evaluated the performance of their algorithm and furthermore, manually evaluated the results of 144 differencing scenarios. They could show that their algorithm performs better and is more accurate than related work conducted in this area. Their approach to map source code in a tree-based structure inspired us to map the output of build logs into a similar structure as well. We initially had the idea to repurpose their source code differencing algorithm and apply it on our also tree-based representation of build logs. We later found that the algorithm was not quick enough in calculating the difference of a large number of long log lines, and therefore used a simpler line differencing approach. However, we have retained the idea of mapping a build log in a tree-based domain model.

Breaking Builds In an empirical study, Seo et al. analyzed 26.6 million builds produced by the developers of Google [33]. They focused on Java and C++ builds and found that C++ developers build 10.1 times a day on average, whereas Java developers build 6.98 times a day. Furthermore, they found that 37.4% of the C++ and 29.7% of Java builds fail. The median time to resolve build errors were 5 minutes for C++ builds and 12 minutes for Java builds. The study also found that the most common cause of build errors is associated with dependencies between components. While writing this thesis, we also found that especially dependencies are the cause of many differences and therefore, a possible cause of build failures. BLogDiff is intended to support developers in edge cases, such as changing transitive dependencies. Since the study identifies such changes as one of the main failure causes, one could claim that BLogDiff has a right to exist even beyond just supporting developers in rare cases.

Kerzazi, Khomh, and Adams analyzed 3,214 builds of a large software company in an empirical study [25]. They found a build breakage ration of 17.9% and quantified the cost of the build breakages as more than 336.18 man-hours. Furthermore, they interviewed 28 software engineers of the company in order to understand the circumstances, in which build break. They found that on average, it takes 57 minutes to fix a build. The developers all agreed that working in smaller teams reduces the build breakage ratio, what they could also show in their data. The build breakage ratio also depends on the role within the team. They recorded that people with the role of an integrator had nine build breakages per month, followed by backend developers with eight breakages. The least amount of breakages had people with the role of the technical lead with one break a month. The primary causes for build breakages were transitive dependencies and mistakenly checking in work-in-progress. This study further emphasized the role of transitive dependencies on breaking builds, which allows us to make a stronger claim on the importance of BLogDiffÅdditionally, we could show in the evaluation of our survey that BLogDiff especially helps inexperienced developers to find the failure cause more quickly, and therefore reduce the cost of broken builds in a CI environment, by reducing the time other developers' work is blocked.

Fixing Broken Builds Macho, McIntosh, and Pinzger introduced a tool called BUILDMEDIC, which is an approach to automatically repair Maven builds that break due to dependency related issues [27]. They manually investigated 37 broken Maven build in 23 open-source Java projects. To fix broken builds, they introduced three different automatic repair strategies, particularly the Version Update, the Delete Dependency, and the Add Repository strategy. They evaluated their tool on 84 broken builds and could show that BUILDMEDIC was able to repair 54% of the broken builds automatically.

Zemp developed BART, a tool that summarizes build failure, within the scope of his bachelor's thesis [38], which was later published as a paper by Vassallo, Proksch, Zemp, and Gall [36]. BART is available as a Jenkins plugin that automatically summarizes a broken build and mines for solutions on Strack Overflow. He further conducted a case study involving eight developers and could show that BART was able to reduce the time needed to fix broken builds by 43%.

BLogDiff is intended as an analysis tool, which supports the process of finding build failure causes by highlighting differences to the previous successful counterpart and providing a set of features to read the output of a build log more quickly. BUILDMEDIC, as well as BART, both choose another approach and try to support developers by presenting or applying solutions to broken builds. They are, therefore, a step ahead in the process of resolving build failures by directly proposing solutions. However, BLogDiff is a more generic tool and can be used in a more versatile way.

Chapter 8

Conclusion

Continuous integration is an emerging trend in the software development industry that is becoming the standard on more and more projects [23]. This thesis focuses on the building of software, which is a crucial part of the continuous integration processes. More precisely, it focuses on failing builds. Failing builds can be a problem in such a high-performance environment and as a failed build can block the work of multiple people at once [25]. Often, finding the failure cause in a failed build is a straight forward process. Sometimes though, the failure cause might be embedded in several thousand lines and finding it can be very a very tedious process. This thesis is intended to help a developer in such edge cases. We researched why build logs cannot be compared using ordinary differencing tools, we studied the build log domain and developed a tool including a browser extension, that allowed us to evaluate whether the process of finding failure causes in build logs can be supported.

We found that build logs cannot be compared using ordinary differencing tools since they contain noise. We defined noise as artifacts that do not hold relevant information and changes from execution to execution. We then classified different forms of noise and built a module that allowed us to filter such artifacts. We could show that the average number of changes between two consecutive build logs decreased from 59% without filtering to 13% with filtering.

When researching the characteristics of build logs, we found that a separate similarity threshold for update classification of download lines increases the quality of the differencing drastically. Moreover, the composition of an edit script of two passing build logs differs from the composition of a passing build followed by a failed one primarily in the number of updates and deletions. Additionally, a difference between a passing build log followed by a failed build log contains usually more updates near the beginning, and the end of the build log as two succeeding passing builds.

We exposed our tool to different communities and surveyed their members to evaluate whether a specialized tool can support developers in finding failure causes. We found that BLogDiff is better suited for inexperienced developers. Experienced developers, in general, need less time to find failure causes. Moreover, they are more familiar with the build log domain, and therefore, do not need the support of a specialized tool. The tool is built in a userfriendly way, and provides additional functionality that, even if a developer is experienced enough, can be attractive features to have.

Bibliography

[1] Angular. https://angular.io/. Accessed: 08.07.2019.

[2] BLogDiff - Web application. https://web.blogdiff.net/differencing. Accessed: 15.07.2019.

[3] Computed Diff - Diff Checker. https://www.diffchecker.com/diff. Accessed: 27.04.2019.

[4] Continuous Integration and Delivery - CircleCI. https://circleci.com/. Accessed: 12.07.2019.

[5] Everything DevOps. https://www.reddit.com/r/devops/comments/cbj7xx/i_developed_a_tool_which_might_make_dealing_with/. Accessed: 27.07.2019.

[6] Jenkins. https://jenkins.io/. Accessed: 12.07.2019.

[7] Levenshtein distance - Wikipedia. https://en.wikipedia.org/wiki/Levenshtein_distance. Accessed: 18.07.2019.

[8] List of Populare Open Source Java Build Tools. https://devopscube.com/list-of-popular-open-source-java-build-tools/. Accessed: 12.07.2019.

[9] Maven - Introduction. https://maven.apache.org/what-is-maven.html. Accessed: 12.07.2019.

[10] Maven - Introduction to the Build Lifecycle. https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html. Accessed: 12.07.2019.

[11] Maven - Introduction to the Dependency Mechanism. https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html. Accessed: 12.07.2019.

[12] Maven - Introduction to the POM. https://maven.apache.org/guides/introduction/introduction-to-the-pom.html. Accessed: 12.07.2019.

[13] Spring Boot. https://spring.io/projects/spring-boot. Accessed: 08.07.2019.

[14] SurveyJS: Free Online Survey and Quiz Tools. https://surveyjs.io/. Accessed: 18.07.2019.

[15] The world's leading software development platform · GitHub. https://github.com/. Accessed: 12.07.2019.

[16] Travis CI - Test and Deploy with Confidence. https://travis-ci.com/. Accessed: 12.07.2019.

[17] Travis Forum. https://travis-ci.community/t/i-developed-a-tool-which-might-make-dealing-with-build-logs-easier-and-i-would-like-to-hear-what-you-think/4153/2. Accessed: 27.07.2019.

[18] M. Beller, G. Gousios, and A. Zaidman. Oops, my tests broke the build: An analysis of travis ci builds with github. Technical report, PeerJ Preprints, 2016.

[19] H. N. Boone and D. A. Boone. Analyzing likert data. *Journal of extension*, 50(2):1–5, 2012.

[20] N. Chavannes. Dataset build log differencing. https://doi.org/10.5281/zenodo.3355554, July 2019.

[21] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM.

[22] V. Frick, C. Wedenig, and M. Pinzger. DiffViz: A Diff Algorithm Independent Visualization Tool for Edit Scripts. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 705–709, 2018.

[23] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 426–437. ACM, 2016.

[24] S. Jamieson et al. Likert scales: how to (ab) use them. *Medical education*, 38(12):1217–1218, 2004.

[25] N. Kerzazi, F. Khomh, and B. Adams. Why do automated builds break? an empirical study. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 41–50. IEEE, 2014.

[26] C. Macho, S. McIntosh, and M. Pinzger. Extracting build changes with builddiff. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 368–378, May 2017.

[27] C. Macho, S. McIntosh, and M. Pinzger. Automatically repairing dependency-related build breakage. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 106–117, March 2018.

[28] M. Meyer. Continuous integration and its tools. *IEEE Software*, 31(3):14–16, May 2014.

[29] noahch. noahch/blogdiff: Release v1.0. https://doi.org/10.5281/zenodo.3355558, July 2019.

[30] noahch. noahch/blogdiff-web: Release v1.0. https://doi.org/10.5281/zenodo.3355562, July 2019.

[31] noahch. noahch/travisci-buildlog-crawler: Release v1.0. https://doi.org/10.5281/zenodo.3355560, July 2019.

[32] T. Rausch, W. Hummer, P. Leitner, and S. Schulte. 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*, pages 345–355. IEEE Press, 2017.

[33] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge. Programmers' build errors: A case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 724–734, New York, NY, USA, 2014. ACM.

[34] M. Sulír and J. Porubän. A quantitative study of java software buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 17–25. ACM, 2016.

[35] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 805–816. ACM, 2015.

[36] C. Vassallo, S. Proksch, T. Zemp, and H. C. Gall. Un-break my build: assisting developers with build repair hints. In *Proceedings of the 26th Conference on Program Comprehension*, pages 41–51. ACM, 2018.

[37] Y. Wang, D. J. DeWitt, and J. . Cai. X-diff: an effective change detection algorithm for xml documents. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, pages 519–530, March 2003.

[38] T. Zemp. Bart, build failure summarisation. Bachelor's thesis, University of Zurich, 2017.