Master Thesis

July 28, 2020

Automatically repairing environmental build failures

Emirald Mateli

of Përmet, Albania (16-726-804)

supervised by

Prof. Dr. Harald C. Gall

Carmine Vassallo

Master Thesis

Automatically repairing environmental build failures

Emirald Mateli

University of Zurich UZH

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

Master Thesis

Author: Emirald Mateli, emirald.mateli@uzh.ch

Project period: 28 Jan 2020 - 28 Jul 2020

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

Acknowledgements

I would like to thank Prof. Dr. Harald C. Gall and the Software Evolution and Architecture Lab of the University of Zurich for giving me the opportunity to conduct this work in their group. I would also like to thank my supervisor, Carmine Vassallo for the guidance, inputs, and expertise shared with me during the work of this thesis. Special thanks go to Christian Macho and Selene Lobnig from the University of Klagenfurt, for their assistance and inputs with the inspection process and the repair strategies conducted in this thesis.

Abstract

Continuous Integration is a widely-used software engineering practice in both industry and open-source projects to automate compilation, testing, and quality assurance tasks. Recent studies reveal that troubleshooting build failures is the main barrier that developers encounter when adopting CI. Because of their complexity, developers usually spend at least one hour per day in fixing build failures. While the majority of build failures are caused by expected human mistakes such as the wrong implementation of a method, a non-negligible part of failures (33%) are caused by environmental factors such as flakiness of the build infrastructure. In this thesis, we want to understand how developers fix environmental failures and the extent to which they can be automatically repaired. We inspect 380 failed builds belonging to 42 different environmental failure types from 97 open-source projects written in Java and Ruby and built on Travis CI. Based on the analysis of the resolution patterns of these failures, we devise and implement an approach for automatically repairing 10 environmental build failure types. To show the applicability of our approach, we run our tool against 67 environmental build failures from popular GitHub projects achieving an overall success rate of 55.22%. To assess the usefulness of our automatic repair, we successfully fix 37 builds from GitHub projects and open issues on these projects where we propose to accept the generated patches for those failures. 66.6% agree with the proposed fixes and are willing to use our tool.

Zusammenfassung

Continuous Integration ist eine sowohl in der Industrie als auch in Open-Source-Projekten weit verbreitete Praxis der Softwareentwicklung zur Automatisierung von Kompilierungs-, Test- und Qualitätssicherungsaufgaben. Jüngste Studien zeigen, dass die Fehlerbehebung bei Build-Fehlern das Haupthindernis ist, auf das Entwickler bei der Einführung von CI stoßen. Aufgrund ihrer Komplexität verbringen Entwickler in der Regel mindestens eine Stunde pro Tag mit der Behebung von Buildfehlern. Während die Mehrzahl der Buildfehler durch zu erwartende menschliche Fehler wie die falsche Implementierung einer Methode verursacht wird, ist ein nicht zu vernachlässigender Teil der Fehler (33%) auf Umgebungsfaktoren wie die Flakheit der Build-Infrastruktur zurückzuführen. In dieser Arbeit wollen wir verstehen, wie Entwickler Umgebungsfehler beheben und inwieweit sie automatisch repariert werden können. Wir untersuchen 380 fehlgeschlagene Builds aus 97 Open-Source-Projekten, die in Java und Ruby geschrieben wurden und auf Travis CI basieren, die zu 42 verschiedenen Arten von Umgebungsfehlern gehören. Auf der Grundlage der Analyse der Lösungsmuster dieser Ausfälle entwickeln und implementieren wir einen Ansatz für die automatische Reparatur von 10 umgebungsbedingten Buildfehlertypen. Um die Anwendbarkeit unseres Ansatzes zu zeigen, lassen wir unser Tool mit 67 umgebungsbedingten Um die Nützlichkeit unserer automatischen Reparatur zu beurteilen, reparieren wir 37 Builds aus GitHub-Projekten automatisch, und offene Fragen zu diesen Projekten, bei denen wir vorschlagen, die generierten Patches für diese Fehler zu akzeptieren. 66.6% stimmen mit den vorgeschlagenen Korrekturen überein und sind bereit, unser Tool zu verwenden.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Continuous Integration (CI) is a software development practice that facilitates automatic compilation, testing, and deployment of software artifacts triggered by changes committed into a source code repository [IZ17, HNT$^+$17]. CI enables teams to reduce their release cycle improving overall product quality and customer satisfaction by allowing them to perform multiple integrations and releases in a day [Che15, FF06, VSZ$^+$17]. Since its inception as one of the Extreme Programming practices in 1991 [Bec99], it has become a widely adopted practice and the overall number of projects using CI continues to grow for both open-source projects and industry [HNT$^+$17, KKA14, AP18, CH11]. As a result of the more frequent integrations and code review processes, CI enhances and fosters communication within the team, results in faster deliveries, and higher quality artifacts. [VYW$^+$15].

Despite its many advantages, CI provides developers with some new challenges to address. Initial adoption, learning curve, adapting to the new technology, and troubleshooting build breaks are some of the main inhibiting factors [SBO18, HNT$^+$17, KKA14]. After the implementation of a CI system, verification of the correctness of the artifact is bound to build results. Broken builds slow down development teams since it may prevent releases, testing, or developers from continuing with their main work as fixing the build becomes a priority [Vas20, KKA14]. Developers spend at least one hour per day fixing broken builds with edge cases taking up to several days causing project timelines to slip. An effect even more pronounced on remote teams due to communication overhead [KKA14, CH11].

When looking at failed builds, besides due to the correctness checks, such as code quality or tests, they may fail for reasons unrelated to development activities. We categorize failed builds into two groups: "Verification" and "Non-verification" failures. Verification failures happen when the build fails due to compilation, testing, or quality issues. Non-verification failures (also called environmental build failures) happen due to reasons unrelated to development activities in the CI server, such as infrastructure errors, API limits, invalid CI configuration, missing dependencies, or failing while locating third-party resources and come from the environment where builds are generated [GCZH19].

Current research reveals a non-trivial amount of environmental build failures [T. 17, GCZH19, VSZ$^+$17]. Ghaleb *et al.* [GCZH19] shows that 33% of broken builds are affected by environmental errors. Rausch *et al.* reveals that just git clone errors make up to 27% of the breakages in some projects. While there are many studies about CI and build failures [GCZH19, T. 17, HNT$^+$17, VSZ$^+$17], to the best of our knowledge there are no studies that investigate the possibility of automatically repairing environmental build failures. This thesis aims to address the aforementioned gap by analyzing broken builds and resolution patterns from the developers to answer the following research questions:

RQ_1: **How do developers repair environmental build failures?**
To answer this question, we conduct an empirical study consisting of 380 builds to understand how developers address environmental breakages and to assess which categories from the Ghaleb *et al.* taxonomy (Tables 1 and 2 in the Appendix) are suitable for automatic repair. To understand each breakage we inspect several builds from the introduction of the error until its resolution in a method called "Build Chain" described in Section 4.2. We try to understand the cause of the failure as well as the approaches developers took in their fixes. As a result of this inspection, we produce a list of categories that we're able to repair alongside their corresponding strategies.

RQ_2: **To what extent can environmental build failures be automatically repaired?**
To answer this RQ, we build a prototype tool to automatically repair environmental failures based on strategies devised from RQ_1. We evaluate it using a dataset of broken builds from Java and Ruby GitHub projects with more than 500 stars that use Travis CI [1] during the May-June 2020 period. Our tool is able to repair 55.22% of builds. In a qualitative survey conducted with developers from successfully repaired projects, 66.6% of developers agree with the proposed fix.

[1]https://travis-ci.com

Chapter 2

Background

In this section, we present concepts and ideas needed to understand the rest of the sections in this thesis.

2.1 Package managers

In Linux [1], "package manager" describes a collection of tools to automate the process of installation, removal, and upgrading of software installed on the local machine [BPVPT19]. The package manager downloads files to install, verifies their integrity using file checksums, and performs the installation process according to the instructions contained in the package.

Apt is a package manager for Ubuntu [2], the distribution of choice for Travis CI. The installation process in Apt works by keeping a local copy of the package lists, upgrades as well as packages that exist in the repositories. When an install command is issued, Apt uses the local information to install the package rather than querying directly on the internet. Our inspection results show that not updating the repository information can result in environmental breakages due to the information available locally being outdated.

2.2 Build tools and automation

Projects use build tools to automate the process of compiling the source code into binary, packaging, running automated tests, and sometimes even deploying to remote servers. Often, they perform package management as well. A package manager is a tool that manages external dependencies the project relies upon, such as third-party libraries that are essential for the program to operate. Much like an operating system package manager, but restricted to packages for a specific programming language. These package managers will often query third-party servers for the requested version of the package, download, and automatically install it. In our study, projects are limited to Java and Ruby, thus we encounter two such tools. Maven [3] for Java, and Bundler [4] for Ruby.

Build automation refers to tooling that executes build automation utilities when triggered by an event or in a scheduled fashion. We introduce Travis CI as a continuous integration service that provides build automation to projects.

[1]https://www.kernel.org/
[2]https://ubuntu.com
[3]https://maven.apache.org/
[4]https://bundler.io/

2.3 Travis CI

Travis CI is a continuous integration platform operating at `travis-ci.com` and is one of the many platforms offering CI servers for projects hosted on GitHub. Each build on Travis is a group of jobs that run in sequence. A job is an automatic process that performs the necessary verification checks. Jobs can be necessary to test against multiple environments. A job is also composed of multiple phases, which are steps inside a job. For the purpose of this paper we study only the phases which affect the overall build status: `before_install`, `install`, `before_script`, `script`, `after_script` [5] which also run in this order. Figure 2.1 shows the transitions between the different phases. The install phase takes care of installing any dependencies the job might require. The script phase on the other hand runs the verification process.

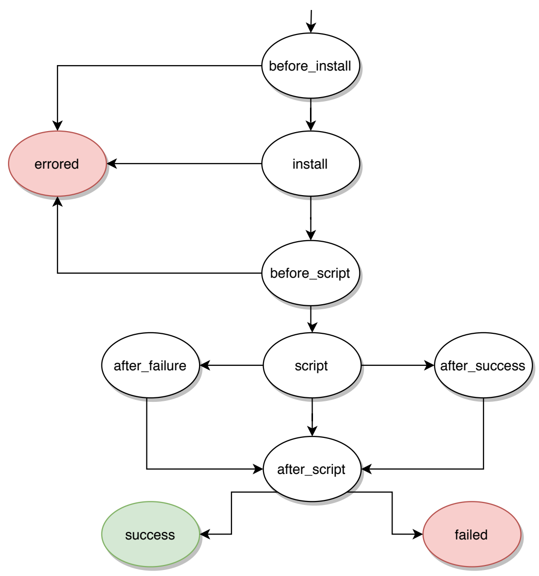


Figure 2.1: Diagram depicting transitions between the Travis CI job phases which affect the build status outcome

A build is finished when all of its jobs are finished. Once a build is finished it is also assigned a label which marks it as passed/failed/errored. The following is a breakdown for each relevant status:

- **Passed**: Exit code 0 returned from all jobs
- **Failed**: Non-zero exit code returned in the `script` phase of any job
- **Errored**: Non-zero exit code returned in one of the following phases of any job:
  - `before_install`
  - `install`
  - `before_script`

[5] https://docs.travis-ci.com/user/for-beginners

Travis is configured via `.travis.yml`, a YAML file in which developers can customize the way build and jobs run. The above sections accept either a string or a list of strings that will be executed by the machine running the job. In this thesis, we will be using these sections to modify the behavior in our efforts to repair the various errors encountered. Travis allows also loading of entire scripts into the build if the behavior is too complex to be contained in a few commands.

Chapter 3

Related work

In related studies, we can find motivating examples of similar work and prior studies on build failures build failure assistance.

3.1 Empirical studies on causes of build failures

In their work, Ghaleb *et al.* study the impact that noise has on breakage data. Current research does not take into consideration environmental failures when studying build breakages, and propose that environmental breakages should not be used to study the association of breakages with development activities [GCZH19]. In their study, Ghaleb *et al.* find that 33% of all build breakages are due to environmental issues. As a result of this study, the paper presents a taxonomy of 11 groups further divided into a total of 61 sub-categories to classify the different environmental errors of 154 Java and Ruby projects presented in Tables 1 and 2 of the Appendix. The work in this thesis is based on this very taxonomy for our environmental error classification purposes. This helps us with studying the resolution patterns of a particular category by inspecting broken builds only from that subset as well as for validating the final repair strategies by testing it against its broken builds.

In their work Rausch *et al.* perform an empirical analysis of build failures in open-source Java projects using a CI workflow. The study focuses on analyzing the types of errors occurring during builds of the studied projects as well as understanding what development practices can be associated with build failures. They distinguish fourteen distinct types of errors occurring in builds. The data is gathered from log files classified with a semi-automatic procedure. Among all categories, the most frequent causes are due to test failures which make up to 80% of the total amount, code rule violations during code inspection, and compilation errors are also prominent in the observations. Their results show that failed builds have non-negligible noise in the data and that errors mostly occur in the first half of the build runtime. In their results, depending on the project, between 9% and up to 27% of failures are caused by git changes not being available to download due to developers merging the changes between the time a build is scheduled and when the job starts [T. 17]. This cause of failure is also present in the Ghaleb *et al.* [GCZH19] dataset, as well as in our validation dataset presented in Section 5.3.

Vassallo *et al.* study types of build failures in open-source and a private company, comparing frequencies of errors between the two. Their research attempts to understand the nature of errors during the build stage in the company and open-source projects. They devise a taxonomy made up of 20 categories that not only encompass verification failures related to compilation, testing, or static analysis but as well as categories such as release, preparation, or deployment [VSZ$^+$17].

We use the results of these works to better understand build failures and their nature. The taxonomy proposed from Ghaleb *et al.* [GCZH19] is the one we will keep using in this thesis moving forward due to its more extensive nature.

3.2 Build failure repair and summarization

Prior studies conducted by Macho *et al.* [MMP18] propose BUILDMEDIC in an effort to automatically repair builds broken due to dependency resolution errors. BUILDMEDIC is able to recognize the dependency error type and apply the appropriate strategy with an overall success rate of 54%. This closely resembles our research goals and proves that successful repair of breakages is possible. The scope of this work lies beyond repairing only of dependency related errors but rather aims to research the feasibility of repair against any sort of environmental breakage by using the previously defined taxonomy from Ghaleb *et al.* [GCZH19] as a cornerstone to study and attempt repair of all 61 sub-categories. If repairing the breakage is not possible, then we want to aid the developers in quickly identifying the cause and pointing to the solution to minimize the time needed to inspect erroring builds which has been found to be non-negligible and may span several workdays [KKA14]. In their study, Macho *et al.* focus on repairing dependency-related breakages whereas, in this thesis, we attempt to repair a number of environmental breakages. In their work Foyzul *et al.* [Has19], empirically assess how to classify build failures into different categories, or a taxonomy similar to other papers [VSZ$^+$17, GCZH19, T. 17]. In addition, they also assess the extent to which build failures can be fixed by altering configuration files [Has19].

Vassallo *et al.* investigate build failures and propose BART, a tool that summarizes the failure in order to help developers understand the key points of the failure and suggest possible solutions found on the internet. Results from this study show that the time to fix a build was reduced on average by 37%, a motivating example for the usefulness of the automatic repair assistance and summarization which lines up with the goals of this thesis [Vas20].

3.3 Program repair

In recent years there have been studies that attempt to automatically repair the failing program. Le Goues *et al.* propose GenProg, a "Genetic Program Repair" which uses existing test cases to automatically generate candidate repair patches which, when applied will cause the tests to pass, signifying "Repair" of the program as defined by Rinard *et al.* [LNFW12, PKL$^+$09]. Ghanbari *et al.* propose PraPR, an automated program repair which aims to help debugging by proposing likely fixes for the encountered bugs by inserting checks in the field references and method calls [Gha19].

These approaches are able to understand the program in order to generate the proposed fixes. According to the definition provided in the introduction, these errors would however fall into verification errors. In this thesis, we aim to repair environmental breakages and thus, do not need to alter or understand the program being tested.

Chapter 4

How do developers repair environmental build failures?

In this chapter, we discuss the approach taken towards answering our first research question. We begin by replicating the results of the Ghaleb *et al.* [GCZH19] study and constructing our initial dataset from the replication results in addition to detailed build information downloaded from the Travis CI REST API. Next, we devise the best way to inspect the builds and describe our method in Section 4.2. To perform the validation, a team of 4 reviewers is set in place inspecting a sample of 380 build chains. Finally, the reviewers, who were split into two groups start a reconciliation process to evaluate and resolve differences in their results. Each group produces two outputs, a list of builds with notes with problem summary, resolution process, suitability for automatic fix, as well as a list of categories that resulted as fixable alongside their corresponding repair strategies.

4.1 Replication of Ghaleb *et al.* study

One of the first steps was to download the replication package provided by Ghaleb *et al.* [GCZH19] and replicate the results of their study as we will be using the build classification tool extensively through this work. As a result of this process, we obtain the original dataset from the study consisting of 35,467 broken builds analyzed and classified according to the breakage type (Table 4.1).

During this process we explore the dataset, randomly inspecting broken builds in an effort to better understand environmental failure and resolution patterns by analyzing the methods and approaches taken by the developers. While the initial dataset proved to be very useful, it was a time-consuming process to find broken builds on the dataset, navigate to Travis CI web UI, GitHub, and all the associated pages. To this extend, we decided to further augment our dataset by downloading the entire build history for the 152 projects. We perform this process through an ad-hoc Python script able to navigate the Travis CI REST API and download build, job, commit information, and raw logs for our data. Besides getting more familiar with the overall taxonomy, the types of errors, as well as knowing how to set up the main inspection procedure, one of the main contributions of this initial process is the decision to inspect build chains rather than individual builds.

Group	Builds	Repositories	Spread
01-Internal CI issues	9,220	150	0.98684
02-Exceeding limits	10,979	142	0.93421
03-Connection issues	8,732	138	0.90789
04-Ruby & bundler issues	2,432	76	0.5
05-Memory & disk issues	1,837	78	0.51316
06-Platform issues	332	53	0.34868
07-Virtual Machine issues	617	54	0.35526
08-Accidental abruption	646	33	0.21711
09-Database (DB) issues	196	2	0.01316
10-Buggy build status	462	41	0.26974
11-External bugs	10	4	0.02632
Total	35,467	152	

Table 4.1: Number of builds organized by group. “Repositories” denotes the total number of projects affected by issues in a particular group, “Spread” is the total ratio against the total number of 152 repositories. In the above data, it can be seen that a few categories such as “09 - Database (DB) issues” or “11 - External bugs” do not affect a large number of repositories.

4.2 Build chains

To better understand the path developers take to fix a build breakage we wanted to avoid cascading breakages, which are builds that are broken due to a previous commit introducing errors. Instead of analyzing builds at random, navigating the builds attempting to figure out the context and the error cause, we wanted to order them in a way that describes a “path” from the introduction of the error until its resolution. To this end, we use the newly downloaded build history from Travis CI to find which builds from our dataset mark the start of a breakage and then to inspect builds from the same branch until the first passed build which signifies that the error has disappeared. It is worth noting that the error may have been resolved in a previous failing build but there was no way for us to automatically know up until what point the error was present. By inspecting until the first passed build, it was guaranteed that the solution would be contained in the chain.

To demonstrate this, in Table 4.2 we present a sample case of a build resolution with a chain of four commits.

Build #	Date	Duration	Status
195	2014-04-01 18:01:31	47 seconds	Errored
228	2014-04-07 18:33:44	1min. 7 seconds	Errored
238	2014-04-08 16:27:40	56 seconds	Errored
246	2014-04-08 20:13:14	27min. 10 seconds	Passed

Table 4.2: Sample build resolution chain for Build #195 in psu-stewardship/scholarsphere in category “03.04 - Server or service unavailable”

In the chain presented in Table 4.2, a commit containing changes performing a version upgrade of a Ruby Gem triggers build 195, which unexpectedly breaks due to an environmental failure: A dependency, `libclamav-dev` all of a sudden fails to install. In the next two commits, the author does not address the issue but rather continues normally with work. Only to be repaired in the final commit with a strategy also recommended by us in our results chapter. The four commits, tell a story and this is what helps us understand the process better, even if in this case the developer decided not to immediately address the error. Inspecting only the second, or third commit without the other two will not be very helpful as we simply have a broken build, without any additional information about the cause or the solution.

In order to construct a build chain, we follow a few steps. The first step is to find the source of the breakage. We consult our dataset of builds that are broken from environmental causes and the result of the previous build is passing. For each of these builds we locate additional builds in the same branch and stop at the first passing build. The branch filter is necessary as it allows us to isolate the current breakage and repair process by not including builds in different branches where different features or fixes are being worked on. An observation can be made that the build chain can also end at the first build belonging in a different category compared to the original one. However, in our inspection process we found that builds may change categories temporarily as the developer while attempting a repair might introduce a new type of error but this still belongs in the same build chain as merely an attempt to repair the original breakage. To demonstrate this idea we show an example in Figure 4.1

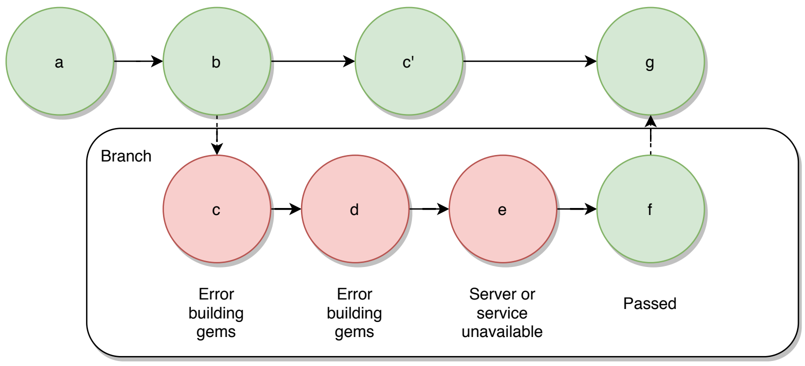
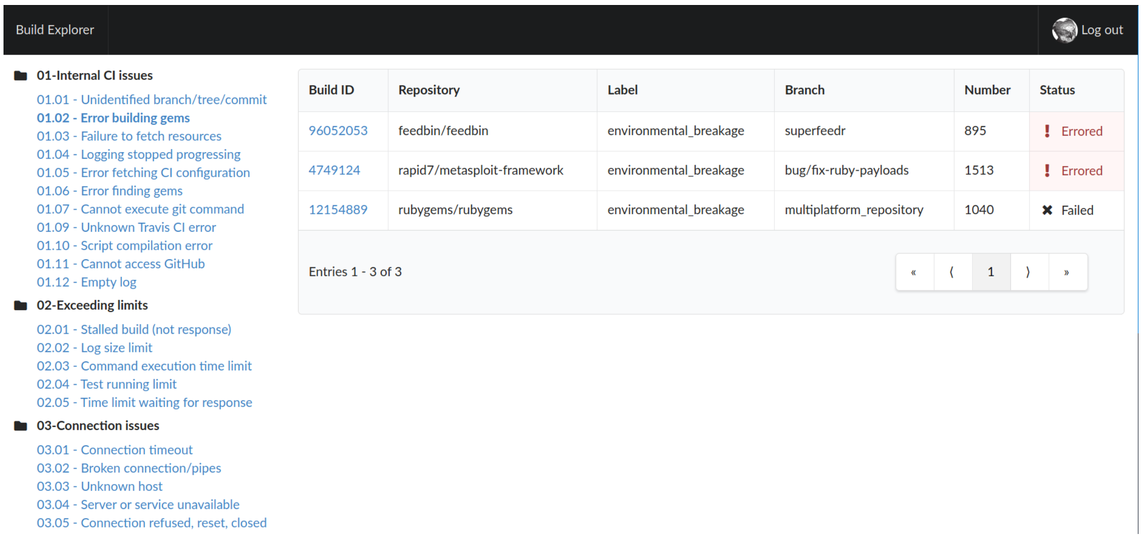


Figure 4.1: Example build chain

In this example, after commit "b" a new branch is created to work on a feature, initially implemented in commit "c", the feature, however, introduces a new dependency that needs to be compiled and introduces the "Error building gems" error due to missing packages in the project. In commit "d" developer unsuccessfully attempts a repair. In commit "e" the developer requests the wrong packages which could not be found and introduces a new type of error. In commit "f", the correct packages are requested and the build process continues normally. This concludes the work done on this branch, which is then merged into the main trunk. In the above example, even though the error category changed, we still considered it as part of the same chain.

4.3 Build inspection platform

For the needs of our analysis, we found out that using the Travis web interface was neither sufficient nor efficient in accomplishing our goals. In addition browsing GitHub, for commits, diff, or pull requests to gather additional context was very often necessary. Although most of this information was already available locally in the datasets generated so far, querying the databases directly was also found out to be not a big improvement. We needed fast navigation between the history of the project, a way to summarize the code changes or errors. To meet our needs an ad-hoc platform was built. Powered by a React [1] front-end, Python [2] back-end running Flask [3], a PostgreSQL [4] instance containing the 35,467 builds, and a MongoDB [5] instance containing the build history of the 152 projects. MongoDB proved quite useful in helping getting started with the data as the JSON documents downloaded from the Travis CI could be inserted into collections without any transformation being necessary and were immediately available for querying. The build listing and navigation are powered by the PostgreSQL data whereas when the user requests detailed build information we use the build history located in MongoDB to provide the full range of details available in the app as shown in Figure 4.2.



01-Internal CI issues
- 01.01 - Unidentified branch/tree/commit
- **01.02 - Error building gems**
- 01.03 - Failure to fetch resources
- 01.04 - Logging stopped progressing
- 01.05 - Error fetching CI configuration
- 01.06 - Error finding gems
- 01.07 - Cannot execute git command
- 01.09 - Unknown Travis CI error
- 01.10 - Script compilation error
- 01.11 - Cannot access GitHub
- 01.12 - Empty log

02-Exceeding limits
- 02.01 - Stalled build (not response)
- 02.02 - Log size limit
- 02.03 - Command execution time limit
- 02.04 - Test running limit
- 02.05 - Time limit waiting for response

03-Connection issues
- 03.01 - Connection timeout
- 03.02 - Broken connection/pipes
- 03.03 - Unknown host
- 03.04 - Server or service unavailable
- 03.05 - Connection refused, reset, closed

Build ID	Repository	Label	Branch	Number	Status
96052053	feedbin/feedbin	environmental_breakage	superfeedr	895	! Errored
4749124	rapid7/metasploit-framework	environmental_breakage	bug/fix-ruby-payloads	1513	! Errored
12154889	rubygems/rubygems	environmental_breakage	multiplatform_repository	1040	✖ Failed

Entries 1 - 3 of 3

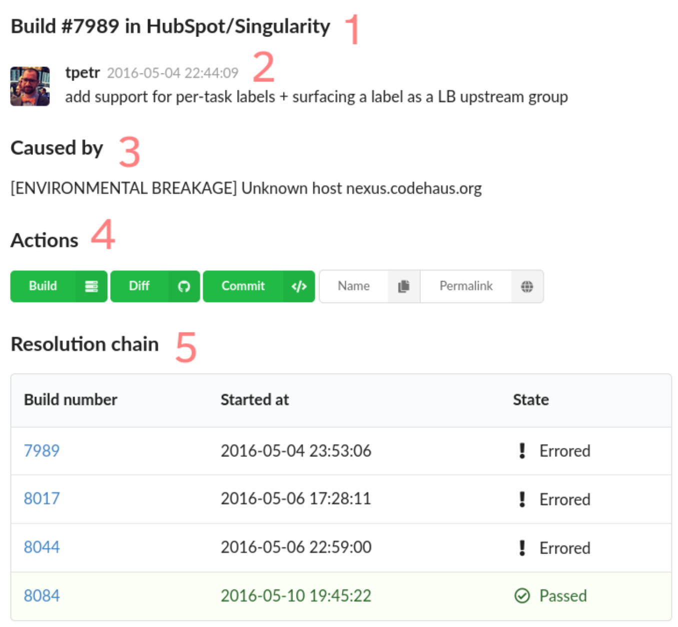
« ‹ 1 › »

Figure 4.2: Build explorer interface showcasing the left navigation menu and builds belonging to a certain category. Screenshot taken from the live environment where each category contains up to 9 builds per category assigned to a reviewer.

In figure 4.3 we showcase different aspects of the "Build details" interface. (1) Build number and repository. (2) Commit information. (3) Cause of breakage as identified by analyzing the build log. (4) Quick links to useful resources related to this diff such as Travis CI, The Git diff, and a link to the GitHub commit page. (5) Table Containing all builds starting from the current one until the one where the breakage is resolved. We notice gaps in build numbers, this is however a normal occurrence as the breakage does not affect the entire repository but rather a git branch. To construct the chain we filter builds only from the branch to which the commit causing the breakage belonged to.

It is at this point that we decide to begin with the inspection process. We have the platform online to facilitate the process, we have an understanding of the failures in our dataset, and how to inspect them via build chains. Next, we decide on how to sample a part of the dataset for inspection.

[1]https://reactjs.org/
[2]https://www.python.org/
[3]https://flask.palletsprojects.com/
[4]https://www.postgresql.org/
[5]https://www.mongodb.com/


Build #7989 in HubSpot/Singularity 1

tpetr 2016-05-04 22:44:09 2
add support for per-task labels + surfacing a label as a LB upstream group

Caused by 3

[ENVIRONMENTAL BREAKAGE] Unknown host nexus.codehaus.org

Actions 4

Resolution chain 5

Build number	Started at	State
7989	2016-05-04 23:53:06	! Errored
8017	2016-05-06 17:28:11	! Errored
8044	2016-05-06 22:59:00	! Errored
8084	2016-05-10 19:45:22	✓ Passed

Figure 4.3: A detail from the build view showcasing a build chain alongside with breakage information and useful links.

4.4 Sample generation

Before calculating the final sample size, the first step was to exclude categories that have a very low spread and the errors happen in a few repositories, which may indicate problems with the testing/CI process in those particular repositories rather than a general problem. To this end, for each category, the number of repositories it affects was also calculated. Categories belonging to the lower quartile were discarded. This left us with a total of 42 categories to inspect. To generate the final sample, a confidence level of 95% with a 5% interval was chosen, yielding the final sample population of 380 builds from 42 categories. This left us with 9 per category builds to be inspected.

A secondary dataset containing only the head of the “build chains” was constructed. These builds are more likely to contain the solution since in reality they represent a set of builds rather than a single one and allows the reviewers to better understand the process that went into solving the build at hand.

The sampling generation algorithm prioritized builds from this new dataset as well as making sure that the builds belong to as many different repositories as possible in order to reduce bias from inspecting only builds from a few repositories, this way more approaches were to be seen. This process was executed for each of the 42 distinct categories to yield the final sample population of builds to be inspected.

4.5 Inspection process

The inspection was carried out by a group of 4 reviewers, facilitated by the platform specifically designed for this process in order to allow easy inspection of build chains. The dataset was randomly split in half, with each half being assigned to two different reviewers, thus creating two pairs. In order to reduce bias, we keep the inspection results private while the process is ongoing.

Each reviewer was assigned several tasks: Manually inspect each of the builds, including following the resolution chain as one of the main tasks to be carried out, find causes for failure, and possible solutions by reviewing the developers' solution or by using information available on the internet. For each build fill out an information sheet containing the following information: `Build "name"`, `Summary of failure`, `Observations`, `Suitable for automated fix?`, `Remarks`, `Links`, `Tags`, `Fixed by developer?` (Figure 4.4).

When both reviewers from a pair would finish the inspection process, they begin a reconciliation process by checking their results and finding a common solution for builds which their repair strategy or suitability for automatic repair differs. Once both reviewers had a common set of results after the reconciliation step, for each repairable category they develop a list of possible fixes alongside concrete strategies. The other pair of reviewers would then review this list, their suitability for automatic repair as well as whether the repair is feasible.

- **Build name**: rspec/rspec-expectations #764
- **Category**: Unidentified branch/tree/commit
- **Tags**: Fixable
- **Fixed by developer?**: No
- **Summary of failure**: Git fails to clone the repository, which causes the job to fail
- **Observations**:
  - Pulling the branch was not possible
  - The branch was deleted (due to pr merge) before it could be cloned
- **Suitable for automated fix?**: Yes
- **Suitable solution**: Recreate the branch based on the pull request diff and re-run
- **Remarks**: None
- **Links**: https://github.com/rspec/rspec-expectations/pull/32

Figure 4.4: A typical build review entry

4.6 Results

This section discusses the results of the build inspection process.

Our build inspection results show 10 categories out of 42 as possible to repair in an automatic fashion, aggregating to 9110 observations, or 25.68% of all the broken builds in the dataset. In the following paragraphs, we give out a summary, insights, and suggestions or solutions for each category separately.

4.6.1 Unidentified branch/tree/commit

Problem summary

The failure occurs when a job starts but the requested git reference, be it a branch, hash, or tag – no longer exists. We identify the following as causes for this behavior:

Case 1: A branch that was requested to be cloned, no longer exists. This commonly occurs with GitHub pull requests, which is the main way for third-party contributors to propose changes to a repository. In this case, the pull request is merged before all the jobs have finished their cloning phase. Upon merging a pull request, GitHub removes the named branch used for that specific pull request, causing the jobs to fail as shown in Listing 4.1. This is also a common occurrence in the study from Rausch *et al.* in which this specific kind of issue ranges from 9% to 27% of the total breakages [T. 17].

Case 2: A branch no longer exists and is unable to be cloned. In this case, the branch was simply deleted and can no longer be referenced.

Case 3: A commit, referenced by its hash, cannot be cloned. In this example, the commit may have been removed by using git's history rewrite operations such as rebasing, amending commits, or simply force-pushing changes onto remote repositories.

Our suggestion

In all three of the above cases the cause of breakage lies with the developers rather than GitHub which only acts as a git service provider, or with Travis. In the case of interacting with pull requests, we recommend developers to wait for build results before continuing with these changes to help them assess that the proposed change meets the projects' standards, especially when changes come from a third-party contributor not directly involved with the project.

If the changes do not affect the system but consists only of textual changes such as documentation then it is possible to skip builds for these kinds of changes as not to add additional overhead for commits, as well as not needing to wait for results which will be known in advance.

```
Cloning into 'spring-cloud/spring-cloud-gcp'...

$ cd spring-cloud/spring-cloud-gcp
$ git fetch origin +refs/pull/2432/merge:

fatal: Couldn't find remote ref refs/pull/2432/merge

The command "eval git fetch origin +refs/pull/2432/merge: " failed. Retrying, 2
     of 3.
```

Listing 4.1: Extract from the errored Build #6159 of spring-cloud/spring-cloud-gcp, due to the developers merging pull request #2432 while the jobs were still being intialized.

4.6.2 Error building gems

Problem summary

The job tries to compile gems(packages/third-party libraries) from source but is unable to do so or is requesting gems that don't exist.

Our solution

(i) Addressing the problem when a developer requests gems that don't exist is something we did not try to resolve, because the name might have been a typo, the version does not exist or perhaps the gem is sitting in a private repository. To this end we let the developers address this issue themselves. (ii) We were able to resolve "building gems" error by including necessary development toolkits such as `make`, `cmake`, `gcc` and the development version of Ruby available in the `ruby-dev` package.

4.6.3 Server or service unavailable

Problem summary

This serves as a catch-all category for HTTP requests performed which return 4XX status codes, indicating bad client requests, 5XX which represent server errors in addition to system package installation failures.

Our solution

Bad requests and internal server errors were deemed as not possible to repair. We were able to address the last issue. In our analysis, we found that developers issued install commands via `apt`, the package manager of choice in Ubuntu distributions that Travis uses. Apt maintains a local copy of URLs that help it locate packages as described in the manpages [6]. Over time, this information might be outdated, and issuing install commands increases the chance that they fail since the URLs might have changed/moved. The more time passes since the last update, the more chances for the error to occur. To avoid this error, it is always recommended to update the repository information before issuing install commands.

4.6.4 Log size limit

Problem summary

Travis CI imposes a log size limit of 4Mb, whenever a job exceeds that limit it will be immediately killed. The two main causes for errors of this kind are: (i) Maven being exceedingly verbose, and (ii) A large number of errors producing big, repeated stack traces.

Our suggestion

To address problem (i) we were able to control and reduce Maven's verbosity and especially trim unnecessary information such as network transfer progress, where Maven repeatedly produces output regarding a downloads' status. This was deemed unnecessary and only transfer errors are shown. Additionally all Maven output below the level `WARN` were removed, this includes log levels such as `DEBUG` and `INFO`. We do not do further modifications so that the rest of the application is unaffected and may still produce `INFO` or `DEBUG` level logs.

Alternative proposals

Another suggestion we considered until the late stages was to embed a new command-line tool in the build pipeline and condense the output. It was observed that several builds in this category

[6]http://manpages.ubuntu.com/manpages/xenial/man8/apt.8.html

have a log which is the same line or lines repeated over and over again, often with the only change being in the timestamp. This was also a very common occurrence with stack traces.

Our proposal was a tool on which Maven output would be piped into and would condense a line or several lines and simply output the line or lines once alongside their frequency. It was decided not to go forward with this implementation however due to not wanting to interfere with the development part of the log by altering the stack-traces or other output.

4.6.5 Wrong build status: Jobs passing but build broken & Build exited successfully

Problem summary

Travis CI reports the wrong build status for builds in these categories. Jobs are passing but the build is marked as broken or the build is marked as passing but any of the jobs were broken.

Our solution

We notify developers via different communication channels of the erroneous status reported by Travis about the build in case.

4.6.6 Flaky categories

As a result of RQ_1 the following categories were found to be repairable. We briefly describe the problem for each of them and offer a similar repair strategy.

Logging stopped progressing

Log gathering for builds in this category stops abruptly and the returned log from the build jobs is only partially retrieved.

Script compilation error

Travis CI errors while executing the `before_script` or `script` phases.

Empty log

Travis CI log gathering fails and the returned log is entirely empty or containing solely the string "null" when retrieved from the API. We have encountered issues containing this bug dating as back as 2012 [7]. A forum post dating from 2017 and persisting up to 2020 shows that the error is still occurring [8].

Time limit waiting for response

The build is waiting for an external resource that is not able to be retrieved in a timely fashion and thus the job is killed.

[7]https://travis-ci.org/github/spree/spree/builds/2331735
[8]https://github.com/travis-ci/travis-ci/issues/7443

Our suggestion

These categories were shown to be also repairable, but we were unable to devise an approach besides re-running. They are mostly internal CI errors or bugs and as such, there is very little agency over them. Due to their flakiness, a re-run is often enough to repair the error.

Chapter 5

To what extent can environmental build failures be automatically repaired?

In this chapter, we present all the steps taken towards answering RQ_2. With the full list of categories and their corresponding repair strategies, we begin the implementation for each of the proposed fixes. We set up a git repository on which all builds will be ran and set up Travis CI to monitor for changes. Once the implementation is finished, we devise ways to test the implementation and create a new validation dataset from broken builds in the May-June 2020 period. After performing a classification process according to the Ghaleb *et al.* [GCZH19] taxonomy, we run build repairs and record the results. We set up a qualitative study by surveying developers responsible for projects which we were able to perform a successful repair for by creating GitHub issues.

5.1 Repair program implementation

In order to test the repair strategies, a tool was built, which, given a repository and a category from the taxonomy, would then attempt to conduct a repair by using the strategies described above. Following is presented the implemented workflow alongside the categories that it was able to repair.

The application is a combination of Bash and Python scripts that accept a Travis CI build ID and a repair strategy that corresponds to a category in the taxonomy. Bash is used for the system calls and fetching the necessary resources while the Python script conducts the actual repair on a repository powered by an ad-hoc library built for specifically interacting with `.travis.yml` files. The process of outputting a commit to trigger a new Travis build from the above input is described below.

Prerequisites: (i) A GitHub repository which will host a mirror of the original repository alongside the newly applied patch. (ii) Travis CI token. (iii) The machine's SSH key added to the GitHub account which owns the mirror repository. (iv) A Travis CI account linked to the GitHub one monitoring the mirror repository for changes.

First, build metadata is extracted using the Travis CI REST API. For our use case, we fetch the repository name and head commit hash. Using this information we are able to download a snapshot of the repository at this specific commit through GitHub. We download the snapshot as a compressed archive. It is not necessary to perform a `git clone` operation as we are not interested in the project history leading up to this point. Using the new files we create a new git repository and add the origin endpoint pointing at GitHub using the `git` protocol allowing us to perform pushes without having to input credentials manually. A necessary step to make the process fully automatic. We run the repair script which accepts a directory containing a git repository alongside the strategy to apply. A commit is created from the modified repository folder and pushed to the git remote. Travis CI will pick up the changes as configured, and trigger a build based on the new commit.

5.2 Repair strategy implementation

Below we describe the implementation for each of the repair strategies.

5.2.1 Unidentified branch/tree/commit

Builds in this category were unable to clone the requested git reference, however, the pull request branch reference no longer exists. To achieve our goal, we reconstruct the Git branch that was meant to be ran. Normally git branches cannot be restored unless a developer has a local copy that can be pushed again, however in this case we utilize GitHub's API to recreate the branch. GitHub is able to provide the Git diff for that particular pull request as well as the head commit hash which the pull request was created from. By using these two pieces of information we are able to recreate the branch as follows: (i) Clone the repository at the head commit, (ii) Apply patch via git. After the branch has been recreated it can be pushed again for Travis CI to run.

5.2.2 Server or service unavailable

The subset of errors we were able to address in this category fail due to `apt` package manager installation errors. The local repository containing information where to download the package is outdated. We are to avoid this error by updating the repository information before requesting a dependency installation. The overhead to this approach is minimal as only metadata related to packages is downloaded rather than the package binaries themselves. To this end, we add a new update command to refresh the repositories as the first action to be executed as shown in Listing 5.1. The `apt` commands issued by our repair process are protected with a guard that checks the existence of apt before issuing any commands. This avoids errors with builds on Windows or Mac OS servers which we do not target.

```
def update_apt_repositories(config):
   return yml_edit_section_arr(config,
                         "before_install",
                         safe_run_aptget("sudo apt-get update"),
                         YML_ACTION_PREPEND)
```

Listing 5.1: Injecting an update command before any other action takes place in the build.

5.2.3 Error building gems

For the repair of builds in this category, we identified that the cause of breakage was the absence of development toolkits such as compilers and the Ruby development version. We install these by issuing the commands shown in Listing 5.2. First, we update the repositories to avoid the "Server or service unavailable" environmental error, then we install the required packages. The commands are prepended into the `before_install` section in order to make sure they are the first ones to run before any other user command takes place.

```
before_install:
 - if [ $(command -v apt-get) ]; then sudo apt-get update; fi
 - if [ $(command -v apt-get) ]; then sudo apt-get install -y build-essential
    ruby-dev libgmp-dev; fi
```

Listing 5.2: Extract from an example `.travis.yml` file. We add the installation of necessary packages to compile Ruby Gems as the output of the repair tool execution.

5.2.4 Log size limit

Our approach to limiting log size was through interacting with Maven and forcing it to run in non-interactive mode as well as limiting the output produced during the installation phase which would needlessly produce large amounts of output regarding artifact download status. Maven is configurable in several ways, through environment variables, command-line arguments, a file named `.Mavenrc` which are used to modify its behavior. Initially, we used the environment variable `MAVEN_OPTS` to configure the behavior but due to how different repositories are configured we found out that using the command-line arguments is the most reliable way of modifying Maven behavior in a repository-agnostic way. We attempt to find maven commands in the four sections of interest `before_install`, `install`, `before_script` and `script` as shown in Listing 5.3 and append the parameters at the end of the command line.

```
def Maven_reduce_verbosity(config):
   mvn_args = "-B -Dorg.slf4j.simpleLogger.log.org.apache.Maven.cli"
         ".transfer.Slf4jMavenTransferListener=warn"

   # check for Maven commands in all sections
   for section in ["before_install", "install", "before_script", "script"]:
      if section in config:
         if isinstance(config[section], str):
            if "mvn " in config[section]:
               config[section] += " " + mvn_args
         elif isinstance(config[section], list):
            for i, command in enumerate(config[section]):
               if "mvn " in command:
                  config[section][i] += " " + mvn_args
   return config
```

Listing 5.3: Injecting Maven command-line arguments to suppress network transfer progress

The above process has a pitfall in that it will fail to modify the arguments for some specific cases such as Maven commands being executed in another script file. However, we observe that the occurrence was not too frequent in our dataset.

5.3 Validation procedure

In this section, we present the approach taken for our validation procedure and generation of the validation dataset used to test the repair rate of the developed strategies and tool. In order to validate our results, we create a new dataset from recent breakages from popular and active projects. This process had also in mind the generation of reports which will be sent to developers for suggestion and approval (Shown in Section 5.5), hence the need to repair breakages that were relatively recent.

The data is gathered from 2462 real-world Java and Ruby open source projects hosted on GitHub which also employ Travis CI. To find mature projects we restrict the number of GitHub stars to at least five hundred. We also limit the languages to Java and Ruby so that they match our original dataset. Since Travis does not have a public catalog of repositories GitHub was used as the main data source to search for projects with the following criteria:

- Language: Java or Ruby
- Stars: ≥ 500

To generate the dataset we need the full list of projects from GitHub. However, GitHub's API endpoint is significantly throttled in terms of call frequency(30 requests per minute down from the normal 83 per minute) and queries return only the first 1000 results regardless of the total number.

To avoid these limits we use the following techniques to generate the dataset:

- Keep the number of requests below 30 per minute by intentionally waiting between network requests.
- Bypass the maximum of 1000 results per query by further segmenting the query by appending GitHub "stars" to the criteria in an automatic fashion as follows:
  - `language:java language:ruby stars:500..519`
  - `language:java language:ruby stars:520..539`
  - `...`
  - `language:java language:ruby stars:2500..2999`
  - `...`
  - `language:java language:ruby stars:7000..40000`

The higher the number of stars, the fewer repositories exist so the stars range can be safely increased without hitting the 1000 results per query. By using this process we were able to download a full list of all 6345 GitHub projects. The next step was to filter out projects not using Travis CI. This was done by first fetching the default Git branch each project uses, since not every project uses a "master" branch, then checking in the tree for the existence of a `.travis.yml` file. The total number of projects using Travis as their CI of choice totaled at 2462, or 38.8%

To finalize the dataset generation process we downloaded the build history using the Travis REST API using a custom-built set of scripts that query, navigate, and fetch all related build and job information. The entire history contained 2,288,926 builds. To limit the amount of data we decided to query broken builds between May and June 2020. Reason for this being that we wanted to test on new data as well as to communicate with the developers with reports sent out in the form of GitHub issues for our qualitative study used for the validation process.

For each build, we only consider jobs which exited without success then download the raw logs to be analyzed and classified.

This process produced 62734 log files downloaded from Travis which we classify using the tools provided from Ghaleb *et al.* replication package [GCZH19]. The distribution according to the environmental issues taxonomy is presented in Table 5.3.

We also compare this with the distribution of the original dataset in Figure 5.1. We note that our dataset contains a higher percentage of builds broken classified as "Unidentified branch/tree/-commit". This discrepancy may come due to a few select projects in our dataset containing a majority of the broken builds in this category. “Unidentified branch/tree/commit” contains 287 projects, but the top four: “activemerchant/active_merchant”, “BetterErrors/better_errors”, “apache/incubator-iotdb”, “apache/druid”, make up 23.917% of the errors.

We also experience lower rates of builds being killed by Travis CI due to exceeding limits. Noteworthy is the (almost) absence of categories such as “Memory & Disk issues”, “Platform Issues”, “Virtual Machine” issues, “Accidental abruption”. From our inspection process, these categories are mostly internal Travis CI errors. Their lack of presence could be due to the possible fixes and improvements in the platform.

Category	**Builds**
01.01-Unidentified branch/tree/commit	5,958
01.02-Error building gems	10
01.03-Failure to fetch resources	362
01.04-Logging stopped progressing	15,362
01.08-Multithreading issues	4
01.11-Cannot access GitHub	11
01.12-Empty log	1
01.13-Caching problems	1
02.01-Stalled build (not response)	925
02.02-Log size limit	285
02.03-Command execution time limit	243
02.06-Job runtime limit	745
02.07-API rate limit	1
03.01-Connection timeout	176
03.04-Server or service unavailable	10,611
03.05-Connection refused, reset, closed	1,276
03.06-Connection credentials error	663
03.07-Remote end hung up unexpectedly	5
03.08-Network transmission error	1
03.11-SSL certificate error	18
04.01-No compatible gem versions	387
04.02-Cannot find, parse, execute gems	106
04.03-Command loading failure	28
04.04-Bad file descriptor	4
04.06-Bundler not installed	318
05.01-Out of memory/disk space	89
05.02-Core dump problems	1
05.03-Segmentation fault	7
06.01-Language installation issues	2
07.01-Improper VM shut down	3
07.02-VM creation error	3
08.01-Build crashes unexpectedly	24
09.02-DB connection error	1
10.02-Build exited successfully	504
Total	38,135

Table 5.1: Distribution of the new dataset in the taxonomy. The much higher representation of category 01.04 appears due to a change in the handling of empty log files.

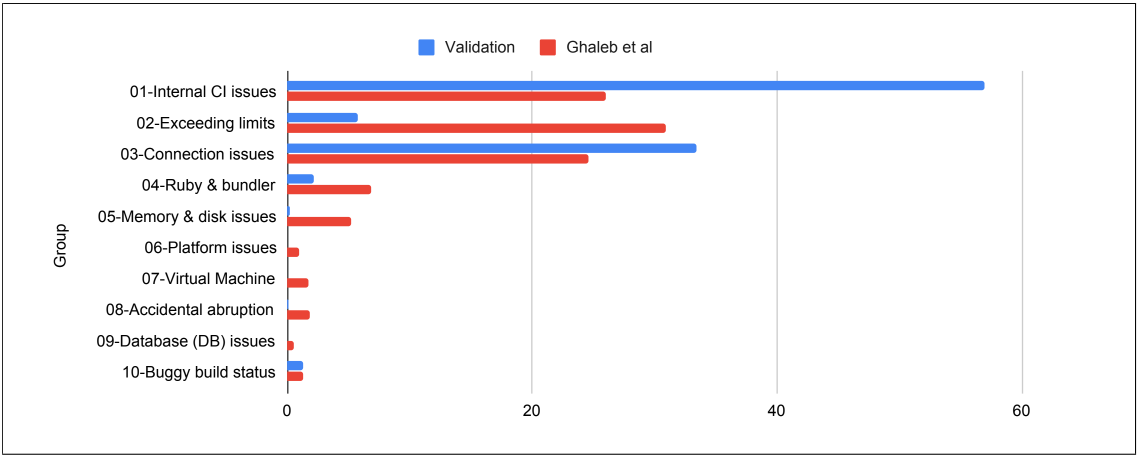


Figure 5.1: Distribution of breakages by build. New validation dataset vs the Ghaleb *et al.* data

5.4 Quantitative study results

In this section, we present the results of our automated repair. We run the prototype against 67 broken builds, limiting to the most recent build per project/category combination.

Builds from the category "01.02 - Error building gems" and "02.02 - Log size limit" were also fetched from the original dataset provided by Ghaleb *et al.* [GCZH19] , we also limit the execution of category "02.02 - Log size limit" to projects using Maven as it is what our solution targets. As the goal was to replicate the original build environment as closely as possible, a new option was added in the build run process to control the Linux distribution used for building. On Travis CI the current Linux distribution used for builds is Ubuntu "Xenial", however, some of our builds are ran with Ubuntu "Trusty" and with older JDK versions which appeared to error during the installation phase from Travis CI due to them no longer being supported. For this reason, we add and use the `use-dist` flag and set it to `trusty` for the older builds. An example of a repair is shown in Listing 5.4.

Category "03.04-Server or service unavailable" which as we have discussed so far, contains many distinct problems we limit our repairs only to builds containing package installation commands in the `before_install`, `install`, `before_script` or `script` sections of the `.travis.yml` configuration file.

Our results yield an overall success rate of 55.22%. The data is presented in Figure 5.2 and Table 5.2. For each of the categories, we include a section where the results are briefly discussed.

```
./runrepair maven-verbosity 687064030

# Inner command executed to conduct the actual repair from above command
./repair \
  --work-dir ~/repairs \
  --strategy "$1" \
  --repository git@github.com:$mirror.git \
  --ref master
  --use-dist trusty
```

Listing 5.4: Repairing build 687064030. Distribution Ubuntu “Trusty” is used, a downgrade from the current “Xenial” release in order to match the original build environment.

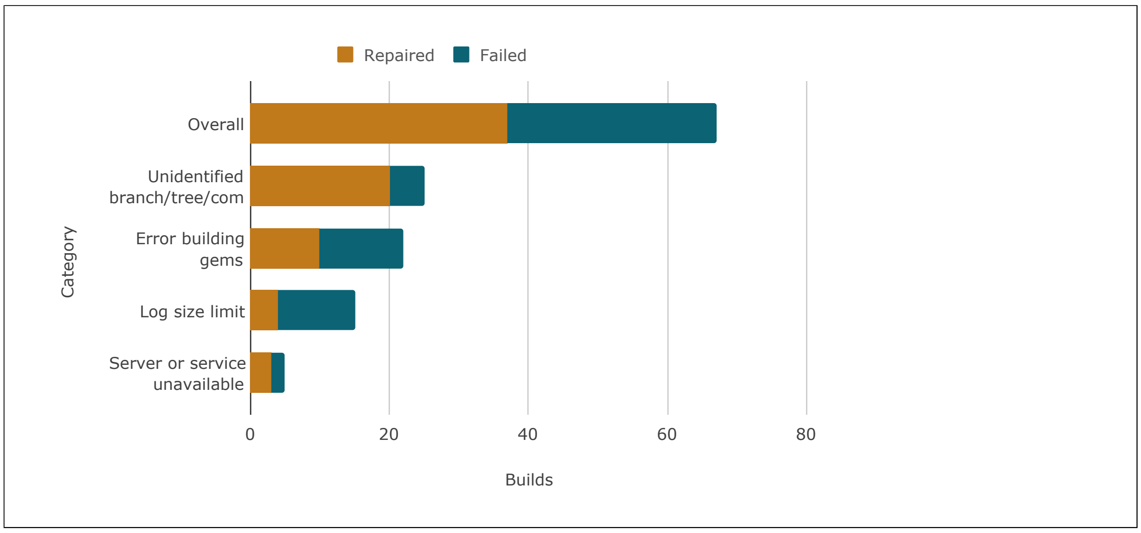


Figure 5.2: Repair statistics by category

Category	Repaired	Failed	Total	Success rate
Overall	37	30	67	55.22%
01.01 - Unidentified branch/tree/commit	20	5	25	80.00%
01.02 - Error building gems	10	12	22	45.45%
02.02 - Log size limit	4	11	15	26.67%
03.04 - Server or service unavailable	3	2	5	60.00%

Table 5.2: Results of repair statistics. Failed denotes builds that were unable to be repaired.

5.4.1 Unidentified branch/tree/commit

Category "Unidentified branch/tree/commit" in practice has a 100% success rate taking into account the goal of the repair is to get the build running regardless of its result, which is always possible. The failures in our results set come from broken builds which were not part of a pull request. The resulting error comes from either a branch being deleted while the build was initializing or any git operation which performs a history rewrite, potentially causing commits to be no longer available, because they were either removed, squashed via rebasing, etc.

Name	**Category**	**Success**
activemerchant/active_merchant #699426008	01.01 - Unidentified branch/tree/commit	Yes
activerecord-hackery/ransack #692962875	01.01 - Unidentified branch/tree/commit	Yes
adomokos/light-service #693975633	01.01 - Unidentified branch/tree/commit	Yes
aidewoode/black_candy #693137940	01.01 - Unidentified branch/tree/commit	Yes
airbnb/synapse #699464415	01.01 - Unidentified branch/tree/commit	No
airsonic/airsonic #690562827	01.01 - Unidentified branch/tree/commit	No
alibaba/arthas #689981989	01.01 - Unidentified branch/tree/commit	Yes
alibaba/easyexcel #698948869	01.01 - Unidentified branch/tree/commit	Yes
alibaba/fastjson #687392372	01.01 - Unidentified branch/tree/commit	Yes
alibaba/jvm-sandbox-repeater #684611569	01.01 - Unidentified branch/tree/commit	Yes
alibaba/nacos #698446758	01.01 - Unidentified branch/tree/commit	Yes
aliyun/aliyun-openapi-java-sdk #696724305	01.01 - Unidentified branch/tree/commit	Yes
oshi/oshi #693060692	01.01 - Unidentified branch/tree/commit	No
gauravk95/bubble-navigation #690228410	01.01 - Unidentified branch/tree/commit	Yes
chef-cookbooks/docker #696829061	01.01 - Unidentified branch/tree/commit	Yes
pravega/pravega #696708758	01.01 - Unidentified branch/tree/commit	Yes
airbnb/synapse #699464415	01.01 - Unidentified branch/tree/commit	No
jnunemaker/flipper #697755516	01.01 - Unidentified branch/tree/commit	Yes
tronprotocol/java-tron #695045282	01.01 - Unidentified branch/tree/commit	Yes
spring-cloud/spring-cloud-gcp #699377884	01.01 - Unidentified branch/tree/commit	Yes
lsegal/yard #682700120	01.01 - Unidentified branch/tree/commit	Yes
apache/kylin #699153478	01.01 - Unidentified branch/tree/commit	Yes
jruby/jruby #694472963	01.01 - Unidentified branch/tree/commit	Yes
RipMeApp/ripme #689514059	01.01 - Unidentified branch/tree/commit	Yes
Shopify/shopify_app #693728377	01.01 - Unidentified branch/tree/commit	No

Table 5.3: Results of automatic build repair for category "01.01 - Unidentified branch/tree/commit"

5.4.2 Error building gems

We were able to repair several builds in this category by installing the necessary development packages as a general solution. There exist, however many builds which were unable to be repaired. These builds required the installation of additional packages that a few Ruby Gems required. However, we kept using our general-purpose strategy rather than adapting it to individual Gem needs which were deemed to be too specific to a repository rather than an overall solution to the category.

Name	Category	Success
chef/chef #42701252	01.02 - Error building gems	Yes
cloudfoundry/cloud_controller_ng #8245585	01.02 - Error building gems	No
errbit/errbit/jobs/27524839	01.02 - Error building gems	No
expertiza/expertiza #40630769	01.02 - Error building gems	No
fatfreecrm/fat_free_crm #3673679	01.02 - Error building gems	Yes
feedbin/feedbin #126338249	01.02 - Error building gems	No
fluent/fluentd #139084821	01.02 - Error building gems	Yes
fog/fog #12954394	01.02 - Error building gems	No
ging/social_stream #6448036	01.02 - Error building gems	Yes
Growstuff/growstuff #8360936	01.02 - Error building gems	No
guard/guard #12566736	01.02 - Error building gems	No
kmuto/review/jobs/138674605	01.02 - Error building gems	Yes
middleman/middleman/jobs/107173351	01.02 - Error building gems	Yes
nanoc/nanoc/jobs/5417491	01.02 - Error building gems	Yes
neo4jrb/neo4j-core #45387041	01.02 - Error building gems	Yes
openshift/rhc #42876035	01.02 - Error building gems	Yes
opf/openproject/jobs/8650479	01.02 - Error building gems	No
pagseguro/ruby #24539687	01.02 - Error building gems	No
rails-api/active_model_serializers #20480376	01.02 - Error building gems	No
grosser/parallel #699142850	01.02 - Error building gems	Yes
puppetlabs/r10k #685959208	01.02 - Error building gems	No

Table 5.4: Results of automatic build repair for category "01.02 - Error building gems".

5.4.3 Log size limit

Builds in this category were able to be repaired at a rate of $4/15$. We notice that our strategy does indeed work and saves space in the log size which in the four passing cases is enough to allow the build to complete. In the case of the builds that were unable to be repaired, the main cause is repeated stack traces being printed throughout all the test suite, long after the Maven installation process. We observe that the space saved by our repair strategy is immediately filled by even more stack traces being outputted due to repeated errors in the build. We are, however, encouraged from the validation dataset regarding our approach being correct in that many of the projects already have a similar implementation in place. Projects such as "apache/druid", "apache/zeppelin", "apache/shiro", "jan-molak/jenkins-build-monitor-plugin" "debezium/debezium", "hugegraph/hugegraph" had a similar or in some cases identical implementation. We also encountered ad-hoc solutions with developers filtering the output via grep commands rather than using Maven's built-in capabilities to reduce logging. Table 5.5 presents the results of this inspection.

Name	Category	Success
hugegraph/hugegraph #694584460	02.02 - Log size limit	No
geotools/geotools #690963648	02.02 - Log size limit	No
openmrs/openmrs-core #684903024	02.02 - Log size limit	No
mybatis/spring-boot-starter #695575606	02.02 - Log size limit	No
DataSystemsLab/GeoSpark #687975530	02.02 - Log size limit	Yes
spring-projects/spring-petclinic #690087792	02.02 - Log size limit	No
siddhi-io/siddhi #687064030	02.02 - Log size limit	No
ron190/jsql-injection #683484467	02.02 - Log size limit	No
yahoo/elide #689367789	02.02 - Log size limit	No
thingsboard/thingsboard #691775483	02.02 - Log size limit	No
debezium/debezium #698460843	02.02 - Log size limit	Yes
bonigarcia/webdrivermanager #694607560	02.02 - Log size limit	Yes
javaparser/javaparser #685292554	02.02 - Log size limit	No
apache/ignite #677028233	02.02 - Log size limit	Yes
apache/hudi #697533820	02.02 - Log size limit	No

Table 5.5: Results of automatic build repair for category "02.02 - Log size limit".

5.4.4 Server or service unavailable

Builds in this category were able to be repaired at a rate of 3/5 as shown in Table 5.6. We observe that builds in this category which were unable to be repaired, require packages that do not exist or otherwise contain problems in their dependencies. We show in Listing 5.5 an example of a failing build due to an inexistent package. The developer possibly meant to require package `python3-pip`, a package manager for Python 3. The current name used is perhaps a confusion due to the binary name of the package being `pip3` (often also aliased as `pip`).

Name	Category	Success
apache/druid #695860094	03.04 - Server or service unavailable	No
apache/incubator-heron #688899916	03.04 - Server or service unavailable	No
apache/jmeter #692282255	03.04 - Server or service unavailable	Yes
apache/storm #696047598	03.04 - Server or service unavailable	Yes
apache/zeppelin #695717856	03.04 - Server or service unavailable	Yes

Table 5.6: Results of automatic build repair

```
$ if [ $(command -v apt-get) ]; then sudo apt-get update; fi

$ wget -q "https://github.com/bazelbuild/bazel/releases/download/${
    BAZEL_VERSION}/bazel-${BAZEL_VERSION}-installer-linux-x86_64.sh"
$ chmod +x bazel-${BAZEL_VERSION}-installer-linux-x86_64.sh
$ ./bazel-${BAZEL_VERSION}-installer-linux-x86_64.sh --user

$ sudo apt-get install pip3

Reading package lists... Done

Building dependency tree

Reading state information... Done

E: Unable to locate package pip3

The command "sudo apt-get install pip3" failed and exited with 100 during .

Your build has been stopped.
```

Listing 5.5: Despite the repair, installation of package pip3 cannot continue as it does not exist.

5.5 Qualitative study results

In order to validate the applicability of our tool in a real scenario, and the quality of the produced patches, besides the build being fixed, we wanted to gather direct feedback from the contributors in each of the repositories and whether they agree or not with our strategies and the validity of our solution. To this end, we generate textual reports from our builds and contact the contributors through opening GitHub issues such as the sample presented in Figure 5.3.

Our issue template contains five important sections. The opening paragraph serves as an introduction to environmental failures and briefly explains to the developers how they can arise. Next, we describe a type of environmental breakage found in the project repository, alongside with links to the breakage as well as the number of occurrences to stress its importance (*e.g.*, 3 occurrences in the May-June period). Then we provide them with our solution or suggestion on how to handle the problem in a textual manner alongside a sample diff representing the necessary code changes in `.travis.yml`. We provide a disclaimer and lastly, we indicate to the developers how to communicate their approval besides replying with a comment. In our results, we use the reactions as definitive agree/disagree responses.

We open issues on the corresponding projects for each of our successful repairs, limited to one per project/breakage type combination, in order not to flood GitHub issues with our repeated recommendations. In total, we create 22 issues whose results we present as follows. We receive 9 responses out of which, in 6 of them the developers agreed with the proposal, in three issues developers express their lack of concern for the issue presented to them. We acknowledge as positive responses cases when developers react with "thumbs-up" as suggested in the issue, state favorable opinion in the comments, or otherwise mark the issue as a task needed to be done. In total, 66.6% of the developers agree with our proposed fixes. We also observe several cases where the developers are not concerned about the environmental breakages occurring in their projects. We also encounter reactions where the developers are initially not concerned but later agree with our proposal.

The full list of issues and their responses are presented in Table 5.7 and Table 5.8.

Build failures often indicate a fault with the software being built such as compilation or test failures. However, there is a subset of build failures (called *environmental* or *non-verification failures*) that are not expected within a standard application development lifecycle, such as build configuration failures, dependency resolution failures, infrastructure failures, and so on.

Problem: One type of environmental failure is due to due to git references being no longer available at the time a build job starts. For example, pull requests are merged shortly before the start of the scheduled build, causing a build failure due to an unfound (because merged) branch. In your project, we detected 3 failures of this type over the May-June timespan. For instance, we found that Build #20361 did not pass due to the above issue.

Solution: It is generally recommended to wait for the build results before merging a pull request. If you wish to skip builds you may also use the SKIP CI tag in your commit message. Additionally, it is also possible to skip builds containing non-code changes as shown in this example and below:

```
+ before_install:
+ - |
+ if ! git diff --name-only $TRAVIS_COMMIT_RANGE | grep -qvE '(.md)|(.png)
    |(.pdf)|(.html)|^(LICENSE)|^(docs)'
+ then
+ echo "Only doc files were updated, not running the CI."
+ exit
+ fi
```

Disclaimer: I developed a tool that repairs environmental build failures and I am now evaluating its usefulness for open-source projects.

Please up/downvote the issue to indicate whether you agree/disagree with the report and the proposed fix.

Figure 5.3: Sample issue created for the tronprotocol/java-tron project. Maintainer reacted positively, welcoming us to create a pull request with the proposed changes.

Issue	Reaction
1. github.com/activemerchant/active_merchant/issues/3698	
2. github.com/activerecord-hackery/ransack/issues/1139	
3. github.com/adomokos/light-service/issues/198	
4. github.com/aidewoode/black_candy/issues/56	Won't fix
5. github.com/alibaba/arthas/issues/1309	
6. github.com/alibaba/easyexcel/issues/1470	
7. github.com/alibaba/fastjson/issues/3348	
8. github.com/alibaba/jvm-sandbox-repeater/issues/74	
9. github.com/alibaba/nacos/issues/3345	Acknowledged, Agree
10. github.com/aliyun/aliyun-openapi-java-sdk/issues/440	
11. github.com/chef-cookbooks/docker/issues/1126	
12. github.com/gauravk95/bubble-navigation/issues/31	
13. github.com/jnunemaker/flipper/issues/473	Acknowledged, Won't fix
14. github.com/jruby/jruby/issues/6325	Acknowledged, Won't fix
15. github.com/lsegal/yard/issues/1342	
16. github.com/pravega/pravega/issues/4947	Acknowledged, Agree
17. github.com/RipMeApp/ripme/issues/1706	
18. github.com/spring-cloud/spring-cloud-gcp/issues/2465	Acknowledged, Agree
19. github.com/tronprotocol/java-tron/issues/3283	Acknowledged, Agree
20. github.com/grosser/parallel/issues/279	Acknowledged, Agree
21. github.com/bonigarcia/webdrivermanager/issues/513	Acknowledged, Agree
22. github.com/DataSystemsLab/GeoSpark/issues/463	

Table 5.7: Issues created alongside with their noted responses.

Category	Positive responses	Nr. of responses	Acceptance rate
Unidentified branch/tree/commit	4	7	0.57
Error building gems	1	1	1
Log size limit	1	1	1
Overall	6	9	0.66

Table 5.8: Responses groupped by category

Chapter 6

Discussion

6.1 Implications

In this section, we discuss implications as a result of the work done in this thesis.

6.1.1 Implications for developers

Disregarding build results

According to our observations, the over-representation of “Unidentified branch/tree/commit” in our validation dataset, in the Ghaleb *et al.* [GCZH19] study as well as in the study from Rausch *et al.* [T. 17], is that developers simply do not wait for build results before merging new code into the repository and undervalue the CI as a process. We urge developers to wait for the build results in order to make an informed decision about the quality of code being added into the codebase. This is especially true about pull requests are the vast majority of the causes for the "Unidentified branch/tree/commit" category and that come from third-parties rather than current maintainers of the project.

6.1.2 Implications for researchers

Immediate feedback

We argue that the best way to implement the approach suggested in this thesis is in the form of immediate feedback to the developer rather than sending them in bulk or otherwise. As shown in the study from Kerzazi *et al.* [KKA14]. Developers may spend up to several days working towards resolving a build failure. Immediate feedback will give the proper guidance and provide developers with a solution and if that is not possible, then to point them towards the proper method to address this in a similar fashion such as the one described by Vassallo *et al.* [Vas20].

Installation of dependencies

We notice that in several cases, developers refer to apt package names by the name of the installed binary, which is often different from the name of the package. We present an example case from project `apache/incubator-heron` [1] where the package `python3-pip` is requested using the binary name `pip3`. This could be a new possible repair method, in which a strategy is devised to correct this kind of mistake and provide the correct suggestion or a list of possible candidates.

[1]https://travis-ci.org/github/apache/incubator-heron/builds/688899916

Categories are too general

A few categories are too general and encompass a very wide range of issues, such as "03.04 - Server or service unavailable" which encompasses all sorts of network failures. Http errors returning 4XX, 5XX status as well as failure to install system packages are bundled here. However, these three things represent very distinct cases that need to be handled differently. Http 4XX are client errors, meaning the client is incorrectly performing the request. Http 5XX are server errors, in this case, the client is no longer to blame, but rather the server has encountered an error or is otherwise misbehaving. Installation of system packages is an entirely different thing that shares very little with Http requests, besides it being the transport protocol.

Language specific categories

Few categories are language-specific, while the study was performed using only Java and Ruby projects, the taxonomy is better served by staying language agnostic. Examples of this include all of the categories under "04-Ruby & bundler issues", it could then be argued that a group of "Java & Maven issues" is also due, or "Java & Gradle issues". We argue that the entire taxonomy needs to be several levels deep instead of having just two levels of depth.

Classification method

The way a build is classified is through matching it against several regular expressions or constant text strings. This approach has the pitfall that it generates too many false positives. Merely including a few strings such as "No output has been received in the last" **anywhere** in the output will incorrectly misclassify it as "02.01-Stalled build (not response)" (sic).

Some example categories suffering from false-positives due to this kind of classification being solely based on substring searching and patterns are the ones from "03 - Connection issues", such as "03.04 - Server or service unavailable" or "03.05 - Connection refused, reset, closed".

In these categories we notice a large number of false positives not necessarily due to external factors, some third-party service being unavailable; but rather services the developers have requested to be installed are not correctly configured, or servers running locally are the ones returning these errors making them wrongly classified as environment breakages as we argue that this is the developers' responsibility to properly configure Redis, Mysql, PostgreSQL, Selenium or other similar services. We show an example of such a case in Figure 6.1.

```
Starting simulator
OpenJDK 64-Bit Server VM warning: Ignoring option MaxPermSize; support was
    removed in 8.0
nc: connect to localhost port 8096 (tcp) failed: Connection refused
```

Figure 6.1: Log extract taken from job 688789747. The test suite was attempting to connect to a local server.

6.1.3 Implications for CI vendors

Caching mechanism for git clone results

One of the most prominent categories in the taxonomy is "Unidentified branch/tree/commit" which as described occurs when a git reference is no longer available. Travis CI supports a multitude of caching mechanisms, but not for git clone. We argue that this one is an important one due to several reasons: (i) It avoids cases where some jobs were able to clone the repository but others not. (ii) It saves bandwidth. (iii) Makes the overall build faster as long as the nodes where the jobs are executed are closer to each other than with GitHub. (iv) Avoids GitHub errors which are frequent enough to be in a popular enough category in "01.11-Cannot access GitHub ". We argue that all of the above benefits come without a downside to the build process.

We communicate our thoughts about this process to Travis CI support and inquire whether this approach has been considered, but has been rejected, or otherwise if they would be open to implementing such a mechanism. It has been revealed to us via an E-Mail communication that the implementation of this caching mechanism is currently not possible due to technical limitations.

Category volatility

In our results, we notice that the rate in which categories appear changed over time. When comparing the distribution of categories between datasets (Figure 5.1) we notice many of the categories which represent internal errors of the Travis CI platform to be significantly less represented, or not at all represented in the new taxonomy. This has the potential implication that errors have been fixed and flakiness by the tool vendor has been reduced. On the other hand, with the introduction of new tools and processes, there might exist a need to further extend the taxonomy with new categories.

Re-running jobs which fail due to internal CI errors

There are a plethora of breakages happening due to internal CI errors. While builds should be reproducible and thus not allowed to be re-run [HF10]. We argue that some jobs can be re-run when the process failed due to CI. In these cases, we will include all VM creation errors where user code has not been executed yet or "01.12-Empty log" which occurs when Travis fails to collect the build logs. This, of course only masks the flakiness of the CI server, but it should allow user builds a chance to execute and lower the amount of noise and failures in builds due to the CI vendors' infrastructure errors.

Infrastructure errors persisting through the years

In both the dataset by Ghaleb *et al.* [GCZH19] , as well as our new dataset from 2020, we notice infrastructure errors persisting through the years. Such a case is "01.12 - Empty log". The earliest instance we could find in our data with a type of this error is from 2012 [2]. This error was still visible in our new data. A GitHub issue [3] is tracking the progress of this defect, which despite attempts to fix it, is still occurring.

6.2 Limitations

In this section, we list some of the possible threats to validity and our solutions to mitigate each of them.

1. During the inspection process, reviewers inspecting a build, or category might be biased towards a solution if others have come to that conclusion before. We mitigate this bias by keeping the results of the inspection separate for all 4 reviewers until the whole inspection procedure is done.

[2]https://travis-ci.org/github/spree/spree/builds/2331735
[3]https://github.com/travis-ci/travis-ci/issues/7443

2. We avoid getting "bad" samples by prioritizing build chains in our sampling strategy. As we have established in previous chapters, build chains give us a better idea of the breakage rather than a singular failing build. To this extent, we prioritize including build chains in our sample rather than picking individual builds.

3. Another point of bias lies in sampling. Since we analyze only a subset of the data, our samples might be biased. To mitigate this we code the sampling procedure in a way such that the maximum number of projects will be present in the final sample. This way we get a variety of issues from different projects operating on many domains and from as many developers as possible.

4. Another threat we wanted to avoid is having to analyze or repair breakages which do not generalize but occur only in a small subset of the projects. To avoid this, we compute the number of projects for which the error appears, we calculate the quartiles and then discard categories belonging in the lower quartile. This way we avoid categories that are dominated by a few select projects. However, we do restrict the usage of language to Java and Ruby to be consistent with the original dataset, and since some of our repair strategies specifically target technologies from these languages such as Maven or Bundler.

6.3 Potential industrial applications and future work

In this section, we discuss possible future work or practical applications of the results with respect to build breakage resolution and monitoring such as cases of "Unidentified branch/tree/-commit"· The system could reconstruct the pull request branch, run the build and notify the developer of its results.

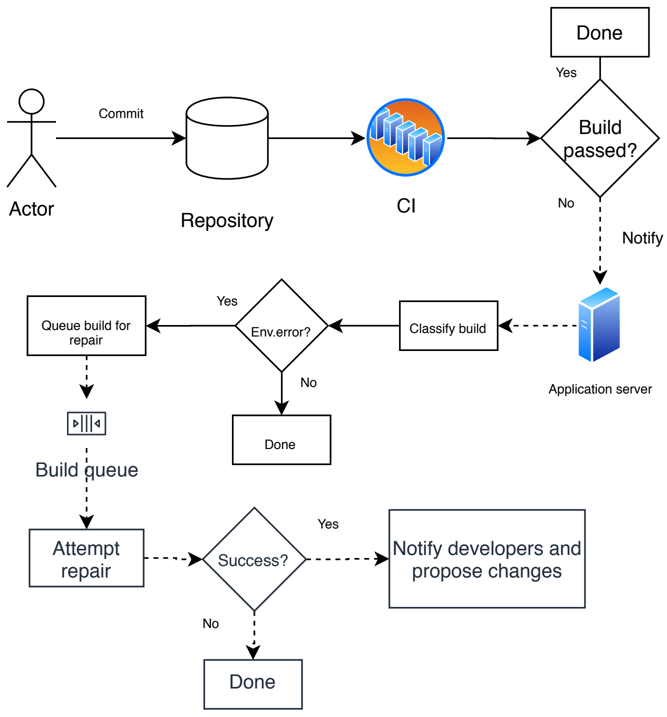


Figure 6.2: Overview of a possible build repair service

The system context in Figure 6.2 gives a high-level overview of a system that is able to provide feedback and propose changes to projects regarding the current failure.

The system is able to be notified by Travis when a certain build fails via hooks. When new builds are received they are run through the build classifier to place the build somewhere in the taxonomy. Unknown errors can be then be manually analyzed in order to extend the taxonomy. Once a build is on a category that is possible to repair it is placed into a queue which workers may periodically poll. Different workers pick builds from the queue and initiate a repair. If the repair was successful then the developers should be notified via channels such as E-Mail, comments on the GitHub project, or pull requests when code changes are proposed which the developers may act upon.

6.4 Conclusions

In this section, we highlight the main contributions made. We inspect 380 builds and build chains to answer the two posed research questions. We propose the use of build chains as a method of inspecting builds in order to get a more complete picture rather than inspecting isolated builds at random. To this end, a "Build explorer" utility was built from scratch and used as the means to facilitate our research goals which we hope researchers interested in studying build failures will use alongside the studying using build chains. A total of 10 out of 42 categories in the taxonomy were found to be repairable. The main limitation lies in that widespread repair success remains a difficult task due to the little agency that exists over many of the causes such as internal CI errors, third-party services being unreliable, slow, or otherwise unresponsive. The results show a repair rate of 55.22% tested against builds on GitHub projects from the May-June 2020 period. In Section 6.3, we propose a concept system making use of the repair strategies and suggesting fixes or otherwise help to the developers while they are addressing the breakage, in an attempt to address one of the barriers of CI adoption [KKA14] by making use of the results of this thesis.

Bibliography

[AP18] S A I B Arachchi and Indika Perera. Continuous integration and continuous delivery pipeline automation for agile software project management. 05 2018.

[Bec99] K. Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.

[BPVPT19] Olivier Bal-Pétré, Pierre Varlez, and Fernando Perez-Tellez. Pacloud: Towards a universal cloud-based linux package manager. In *Proceedings of the 2019 International Communication Engineering and Cloud Computing Conference*, CECCC 2019, page 6–13, New York, NY, USA, 2019. Association for Computing Machinery.

[CH11] M. Cataldo and J. D. Herbsleb. Factors leading to integration failures in global feature-oriented development: an empirical analysis. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 161–170, 2011.

[Che15] Lianping Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32, 03 2015.

[FF06] M. Fowler and M. Foemmel. *Continuous Integration*. 2006.

[GCZH19] Taher Ghaleb, Daniel Costa, Ying Zou, and Ahmed E. Hassan. Studying the impact of noises in build breakage data. *IEEE Transactions on Software Engineering*, pages 1–14, 08 2019.

[Gha19] Ali Ghanbari. Toward practical automatic program repair. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, page 1262–1264. IEEE Press, 2019.

[Has19] Foyzul Hassan. Tackling build failures in continuous integration. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 1242–1245. IEEE Press, 2019.

[HF10] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010.

[HNT$^+$17] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-offs in continuous integration: Assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 197–207, New York, NY, USA, 2017. Association for Computing Machinery.

[IZ17] Md Rakibul Islam and Minhaz F. Zibran. Insights into continuous integration build failures. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, page 467–470. IEEE Press, 2017.

[KKA14] 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, 2014.

[LNFW12] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[MMP18] Christian Macho, Shane McIntosh, and Martin Pinzger. Automatically repairing dependency-related build breakage. In *Proc. of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, page 106–117, 2018.

[PKL$^+$09] Jeff Perkins, Sunghun Kim, Samuel Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Gregory Sullivan, Weng-Fai Wong, Yoav Zibin, Michael Ernst, and Martin Rinard. Automatically patching errors in deployed software. pages 87–102, 01 2009.

[SBO18] Mali Senapathi, Jim Buchan, and Hady Osman. Devops capabilities, practices, and challenges: Insights from a case study. pages 57–67, 06 2018.

[T. 17] 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 (msr 2017). 2017.

[Vas20] Vassallo, Carmine. Proksch, Sebastian. Zemp, Timothy. Gall, Harald C. Every build you break: developer-oriented assistance for build failure resolution. 2020.

[VSZ$^+$17] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. Di Penta, and S. Panichella. A tale of ci build failures: An open source and a financial organization perspective. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 183–193, 2017.

[VYW$^+$15] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. pages 805–816, 08 2015.

Appendix

Category	Builds
01.01-Unidentified branch/tree/commit	4010
01.02-Error building gems	829
01.03-Failure to fetch resources	1893
01.04-Logging stopped progressing	520
01.05-Error fetching CI configuration	1092
01.06-Error finding gems	204
01.07-Cannot execute git command	92
01.08-Multithreading issues	70
01.09-Unknown Travis CI error	79
01.10-Script compilation error	19
01.11-Cannot access GitHub	330
01.12-Empty log	26
01.13-Caching problems	5
01.14-Writing errors	23
01.15-Remote repository corruption	16
01.16-Cannot allocate resources	6
01.17-Storage server offline	2
01.18-Path issues	4
02.01-Stalled build (not response)	5746
02.02-Log size limit	1374
02.03-Command execution time limit	1933
02.04-Test running limit	1549
02.05-Time limit waiting for response	191
02.06-Job runtime limit	157
02.07-API rate limit	29

Table 1: Categories (01-02) and the number of builds present in each. Names verbatim from Ghaleb *et al.* [GCZH19]

Category	Builds
03.01-Connection timeout	2057
03.02-Broken connection/pipes	185
03.03-Unknown host	757
03.04-Server or service unavailable	1675
03.05-Connection refused, reset, closed	2244
03.06-Connection credentials error	1392
03.07-Remote end hung up unexpectedly	220
03.08-Network transmission error	46
03.09-Connection, proxy, & sync errors	43
03.10-SSL connection error	105
03.11-SSL certificate error	8
04.01-No compatible gem versions	662
04.02-Cannot find, parse, execute gems	381
04.03-Command loading failure	722
04.04-Bad file descriptor	253
04.05-Dependency request error	320
04.06-Bundler not installed	94
05.01-Out of memory/disk space	1302
05.02-Core dump problems	158
05.03-Segmentation fault	303
05.04-Memory stack error	72
05.05-Corrupted memory references	2
06.01-Language installation issues	326
06.02-Invalid platform	2
06.03-Unexpected failure	4
07.01-Improper VM shut down	549
07.02-VM creation error	47
07.03-VM connection problem	7
07.04-Invalid VM state	5
07.05-Stalled VM	9
08.01-Build crashes unexpectedly	646
09.01-DB creation quota	159
09.02-DB connection error	37
10.01-Jobs passing but build broken	114
10.02-Build exited successfully	348
11.01-E.g., interpreter bugs	10

Table 2: Categories (03-11) and the number of builds present in each. Names verbatim from Ghaleb *et al.* [GCZH19]